Using Alloy for the Analysis of Model Transformations: A Case Study

Kyriakos Anastasakis and Behzad Bordbar

In Preparation

10 July 2007
(Last Updated: 16 July 2007)

School of Computer Science, The University of Birmingham, UK

CONTENTS

Abstract

In this report we present our experiences on using Alloy for analysing model transformations.

1. INTRODUCTION

The Model Driven Architecture (MDA) [15] aims at promoting the role of models in the software development process. The MDA provides a way to automate transformations between models. A model transformation in the MDA 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. Figure 1.1 depicts an overview of model transformation specifications.

The metamodels of the source and target languages are specified using a common metalanguage, the Meta Object Facility (MOF) [20]. A number of languages have been proposed for the definition of the transformation rules [3, 14] and the Queries/Views/Transformations (QVT) [10] standard specifies the characteristics and capabilities model transformation languages should have.

1.1 Motivation

Like any other kind of software, it is important to be able to apply formal analysis techniques on model transformations to assess their quality. Model transformations can be considered as special kinds of models [7], since they consist of three models, the source metamodel, the target metamodel and the transformation rules. As a result model transformations can be subject to existing model checking and analysis techniques.

A number of methods have been proposed for the analysis of model transformations. A number of methods [26, 17, 16, 5] represent model transformations as graph transformations [24] and use established theories from the domain of graph transformation to prove certain properties, such as uniqueness and termination of the transformation.

Model based testing is also a popular approach used to check model transformations. Model based testing methods import conventional software testing techniques to the domain of Model Driven Engineering (MDE). Those methods [18, 8] propose various techniques to produce a number of test cases, which can then be used as input to model transformations.

In this report we propose the use of Alloy [13] as a language for analysing 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 [13] and has strong foundations on relational logic. *Make this statement more clear...* Similarly some approaches [10, 2] advocate the use of declarative relational techniques to defining model transformations. In particular they use a relational approach to define mappings on the metamodel level in a declarative way.

The next section provides a brief introduction to the Alloy language.

Fig. 1.1: A high level view of Model Transformations

2. ALLOY

Alloy [13] is a textual, declarative modelling language based on first-order relational logic. An Alloy model consists of *Signatures*, *Fields*, *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*.

Each Signature denotes to a set of *Atoms*, which are the basic entities in Alloy. Atoms are *indivisible* (they can not be divided into smaller parts), *immutable* (their properties remain the same over time) and *uninterpreted* (they do not have any inherent properties) [13]. Each Field belongs to a signature and represents a relation between two or more signatures. Such a relation denotes to a set of tuples of Atoms. In Alloy *Facts* are statements, which define constraints on the elements of the model. Parameterised constraints, which are referred to as *Predicates*, can be invoked from within facts or other predicates.

Alloy comes with a tool, Alloy Analyzer [12], 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.

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 [13, Sect. 5].

The following section presents our methodology for the analysis of model transformations via Alloy is presented.

3. DESCRIPTION OF THE APPROACH

Figure 3.1 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.

3.1 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.

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 [22, p. 110]. Such invariants are often referred to as *well-formedness rules* [22]. Well-formedness rules are usually specified using the Object Constraint Language (OCL) [21] 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 [4] that translates MOF metamodels enriched with well-formedness rules expressed in OCL, to Alloy. We have implemented this method in a tool called UML2Alloy [1].

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 [10].

3.2 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* [25] functionality that highlights the statements which lead to logical inconsistencies. This functionality can be used to debug inconsistent model transformation specifications.

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.

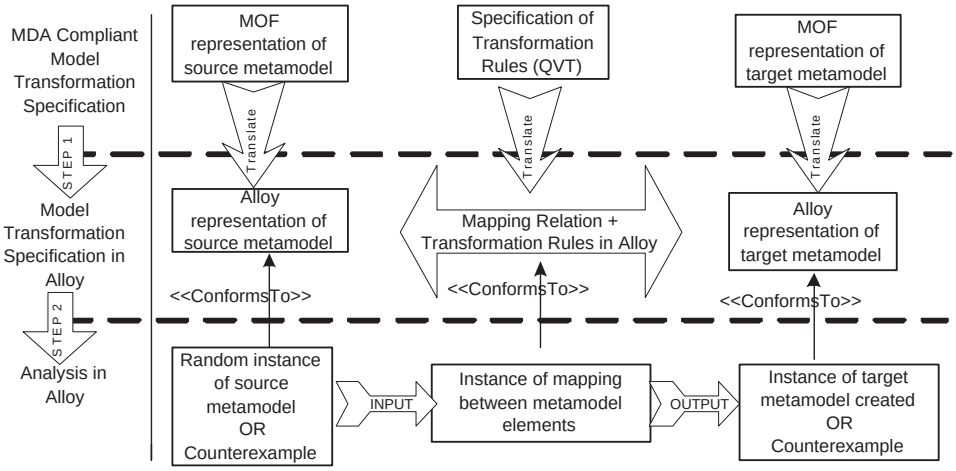

Fig. 3.1: An outline of our approach

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 4 illustrates such a case in more detail.

4. APPLYING OUR APPROACH: A CASE STUDY

Our approach has been applied on a model transformation from the domain of business processes. The next section briefly presents the example transformation. For more details please refer to [18].

4.1 Description of the Example

We have applied our method to the transformation presented in [18] by Küster et al. This transformation deals with a domain specific language used for business process modelling, utilised by IBM's WebSphere Business Modeler [11]. The language is similar to UML Activity Diagrams [23].

The transformation [18] 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 4.1 depicts 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 [18].

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 [18] and as our approach reveals. In the following section we demonstrate how our approach can be applied to the rules depicted in Fig. 4.1.

4.2 Analysis on the Metamodel Level

In order to analyse the model transformation in Alloy we need to first represent the MOF compliant metamodels of the source and target languages as well as the transformation rules in Alloy.

4.2.1 Translating the MOF compliant metamodels to Alloy

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 [18], is depicted in Fig. 4.2.

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 [4]. 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 [4], which implements these rules, we constructed an Alloy representation of the metamodel of the source language.

Figure 4.3 depicts an excerpt of the metamodel, with inline comments. Lines 1 - 9 depict the Signatures of the metamodel and the association ends as represented in the Alloy model. Lines 10 - 14 represent association constraints. In particular they state that the *target* and *incoming* (line 12) as well as the *source* and *outgoing* (line 13) relations are symmetric [13, p. 62]. More specifically the statement in line 12, constraints that the *target* relation will relate

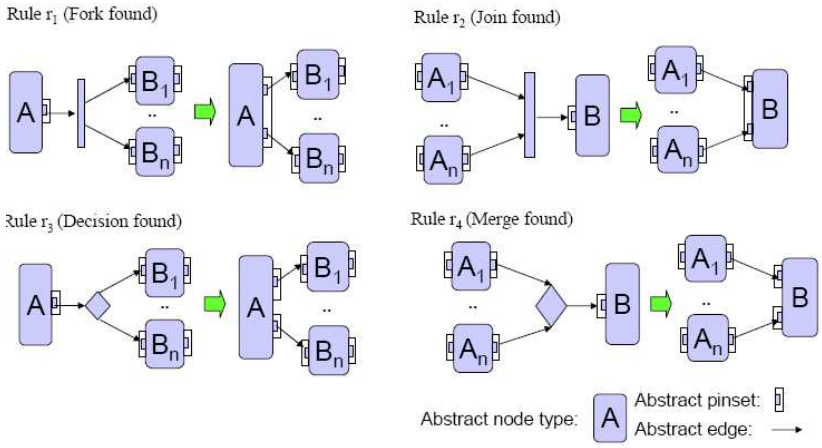

Fig. 4.1: Transformation rules of the CA2PinSet transformation [18]

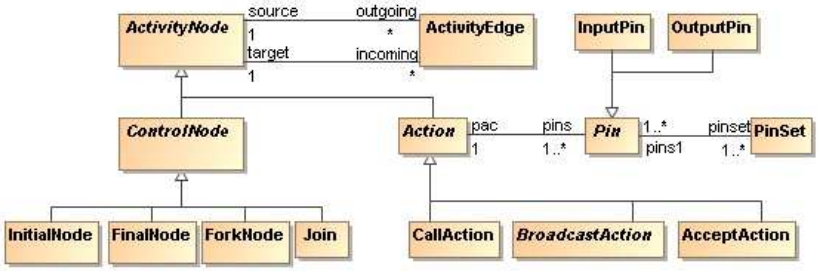

Fig. 4.2: A simplified source metamodel, adapted from [18]

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

4 abstract  sig ActivityNode{ // Represents the ActivityNode Class
5 outgoing: set ActivityEdge, // Represents the 'outgoing' association end
6 incoming: set ActivityEdge // Represents the 'incoming' association end
7 }

8 abstract sig ControlNode extends ActivityNode{} // Represents the ControlNode Class

9 sig JoinNode extends ControlNode{}


10 //Association Facts 11 fact{
12 target = ~incoming
13 source = ~outgoing
14 }

//Multiplicity Facts
15 fact{
16 //The outgoing relation maps one ActivityNode to 0 or more ActivityEdges

17 outgoing in ActivityNode one -> set ActivityEdge
18 incoming in ActivityNode one -> set ActivityEdge
}

19 // Well-formedness Rules
20 fact{
21 // All Initial nodes have 0 incoming edges.
22 all i:InitialNode | #i.incoming = 0
23 }
```

Fig. 4.3: Portion of the transformed metamodel of the source language in Alloy

the same atoms as the *incoming* relation. Similarly the statement in line 13, constraints that the *source* relation will relate the same atoms as the *outgoing* relation.

In fact defining both association ends of an association in Alloy is redundant. The reason for this is that Alloy has no notion of navigable and non navigable association ends. For example whenever the *target* referred to in a statement, it could be replaced by its transpose, namely $\sim$*incoming*. However we defined both the *target* and the *incoming* association ends in the Alloy model, to make the translation from the MOF metamodels to Alloy more natural. For more information on transpose relations in Alloy, please consult to [13, Sect. 3.4.3.4]

Next we need to transform the metamodel of the target language to Alloy.

In the transformation we deal with, 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.

In this transformation the source and target metamodels have elements with the same name (e.g. ActivityNode). Since we represent both the source and target metamodels in one Alloy model, we need to avoid name conflicts (i.e. to distinguish which elements belong to the source metamodel and which elements belong to the target metamodel). As a result the elements of the target metamodel will start with a capital 'T'. Figure 4.4 depicts an excerpt of the target metamodel.

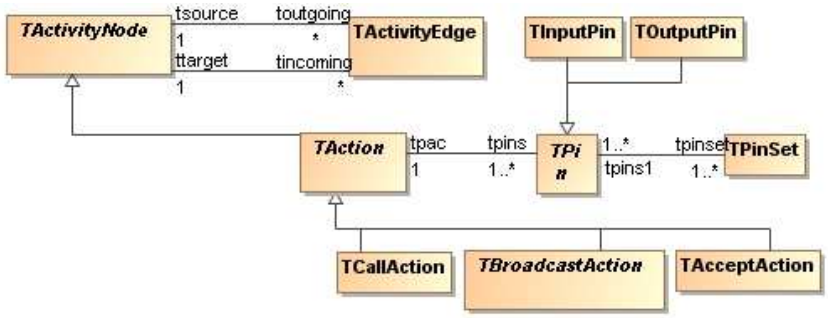

Fig. 4.4: A simplified target metamodel, adapted from [18]

4.2.2 Translating the transformation rules to Alloy

Our approach also requires that the transformation rules are expressed in Alloy. In [18] the transformation rules were expressed using graph transformations. In our approach we need to express the transformation rules using first-order logic statements. We carried out this translation from graph transformations to Alloy manually.

Figure 4.5 depicts an excerpt of the Alloy representation of the transformation rule *r2*, which removes a Join node and replaces it with an implicit join [18].

Lines 1-6 in Fig. 4.5 represent the Mapping relations. The relations comprise of the *Mapping* signature and a number of field declarations that specify which elements of the source metamodel are mapped to which elements in the target metamodel. For example line 2 specifies that an InitialNode of the source metamodel will be mapped to exactly one InitialNode in the target metamodel (*TInitialNode*). Line 3 states that an ActivityEdge will be mapped to at most one ActivityEdge (*TActivityEdge*) in the target metamodel (as shown by *r2* in Fig. 4.1 the outgoing ActivityEdge of a JoinNode does not map to any ActivityEdge after the transformation).

Lines 7-43 depict an excerpt of the rule *r2* defined in Alloy. Lines 7-17 define how an ActivityEdge, which starts from an *OutputPin* and finishes on a *JoinNode* is transformed. Informally this can be described by the following:

For all ActivityEdges of the source model, if the ActivityEdge relates an OutputPin with a JoinNode then execute the outputPin2Join part of the rule (described later), with parameters the ActivityEdge of the source metamodel the ActivityEdge of the target metamodel and the Mapping relation.

Lines 18-33 depict the *outputPin2Join* part of the rule. The *outputPin2Join* part of the rule, describes the details on how the mapping of an ActivityEdge, which starts from an *outputPin* of a Node and finishes at a *JoinNode* takes place. The outputPin2Join rules accepts as parameters the ActivityEdge of the source metamodel the ActivityEdge of the target metamodel and the Mapping relation. Lines 18-24 can be informally described:

Execute the 'outputPin2Any' (line 20), which maps the outputPin of the source metamodel to an outputPin in the target metamodel. Then if the outgoing edge of the Join finishes on a FinalNode (line 22), then the ActivityEdge of the target metamodel (i.e.'tae') is mapped through the mapping relation to the ActivityEdge of the source metamodel (line 23) AND the 'tae' finishes on the Node where the outgoing edge of the original JoinNode finished (line 24).

Figures 4.6 and 4.7 represent the transformation rules using a kind of object diagram. Also show the transformation on the AD.

Figure 4.6 provides a graphical representation of the specifications of lines 22-24 in Fig. 4.5.

Similarly lines 26-33 specify how an ActivityEdge, which starts from an OutputPin and finishes in a JoinNode, whose outgoing edge ends up in an *InputPin* of a *CallAction*. Figure 4.7 depicts a graphical representation of the specification of lines 26-33 in Fig. 4.5.

Looking at the excerpt of the rules in Fig. 4.5 one can observe that the rules map *ActivityEdges* of the source metamodel to *ActivityEdges* of the target metamodel. The rules are different depending on the kinds of elements the edges connect. For example lines 20 and 23 define a different mapping depending on whether the *JoinNode's* outgoing edge finishes on a *FinalNode* or an *OutputPin*. This distinction is necessary due to the way we define

```
1 sig Mapping{
2 in2tin: InitialNode one -> one TInitialNode,
3 ae2tae: ActivityEdge one -> lone TActivityEdge,
4 oup2toup: OutputPin  one -> one TOutputPin,
5 fn2tfn: FinalNode one -> one TFinalNode,
6 }


7 pred mapAE2TAE(){
8 all m:Mapping | all ae:ActivityEdge |
9   (
10       ae.source in OutputPin && ae.target in JoinNode => //OutputPin2Join
11       one tae:TActivityEdge | outputPin2Join[ae,tae,m]
12  ) else
13      (   ae.source in InitialNode && ae.target in InputPin =>
14      (one tae:TActivityEdge | Initial2whatever[ae,tae,m] &&
15                                  any2InputPin[ae,tae,m] )
16      )
17 }



18 pred pred outputPin2Join(ae:ActivityEdge, tae:TActivityEdge, m:Mapping){
19 // Source
20 outputPin2Any[ae,tae,m]
21 //Target
22 ae.target.outgoing.target in FinalNode =>
23 tae = m.ae2tae[ae] &&
24 tae.ttarget = m.fn2tfn[ae.target.outgoing.target]
25 else
26 ae.target.outgoing.target in InputPin =>
27 tae = m.ae2tae[ae] && tae.ttarget = m.ip2tip[ae.target.outgoing.target] &&
28      (
29          all ca:ae.target.pac | ca in CallAction => ( one tc:TCallAction |
30          tc = m.ca2tca[ca] && tc = tae.ttarget .tpac
31          )
32      )
33 }

34 pred outputPin2Any(ae:ActivityEdge, tae:TActivityEdge, m:Mapping){
35 ae.source in OutputPin => (
36   (one top:TOutputPin |
37      top = m.oup2toup[ae.source] && tae.tsource = top)
38      && (
39      all ca:ae.source.pac | ca in CallAction => ( one tc:TCallAction |
40          tc = m.ca2tca[ca] && tc = tae.tsource.tpac  )
41      )
42      )
43 }
```

Fig. 4.5: An excerpt of the mapping relation and the transformation rules

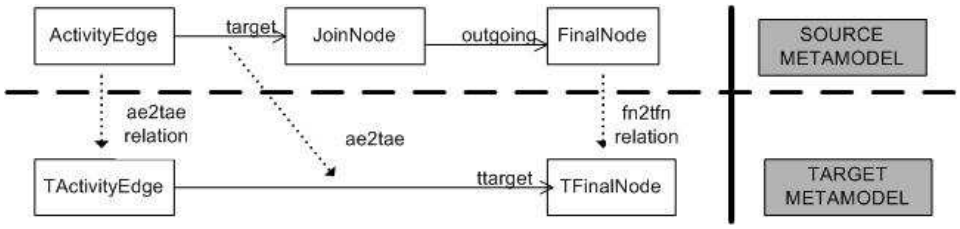

Fig. 4.6: Graphical representation of part of the transformation rules depicted in lines 22-24 in Fig. 4.5

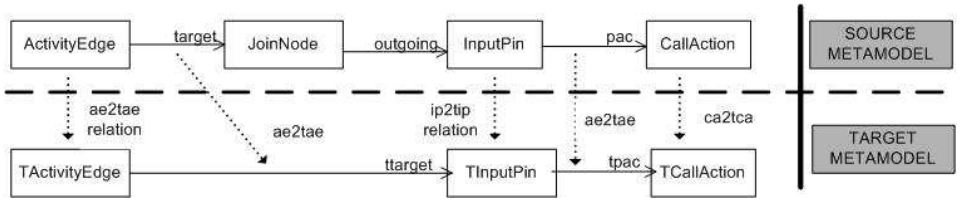

Fig. 4.7: Graphical representation of part of the transformation rules depicted in lines 26-33 in Fig. 4.5

our transformations. More specifically if the outgoing edge of a *JoinNode* finishes on a *FinalNode*, that node will be transformed to a *TFinalNode* though the *fn2tfn* (line 5 of Fig. 4.5) field of the *Mapping* signature. However if it finishes on an *OutputPin* that *OutputPin* will be transformed to a TOutputPin through the *oup2toup* (line 4 of Fig. 4.5) field of the *Mapping* signature.

4.3 Results of the Analysis on the Meta Level

To perform the analysis, Alloy Analyzer was first used to simulate the transformation presented in Fig. 4.1. The Analyzer requires the user to specify a scope [13, 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, 2 CallActions, 4 Pins, 1 JoinNode, 10 ActivityNodes, 2 TActions, 2 TCallActions, 6 TPins, 12 TActivityNodes, 1 Mapping and a default scope of 6. 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.

4.3.1 Simulation

Alloy Analyzer produced an instance of a transformation where a well formed target model was generated. It took the analyzer 3.5 mins to produce the instance for the specified scope.

As mentioned in Sect.řefch:Alloy, the instances in Alloy are represented as sets of tuples of atoms. Figure 4.3.1 depicts an excerpt of the instance of the transformation Alloy Analyzer produced. More specifically this instance represents 5 ActivityEdges, *ActivityEdge[0]*, *ActivityEdge[1]*, etc. It also represents 5 *target* associations. Each association is represented by an ordered pair. One that relates *ActivityEdge[0]* with *JoinNode[0]*, one that relates *ActivityEdge[1]* with *InputPin[0]* etc. Similarly the *source* relation is made of ordered pairs.

This instance was manually converted to the equivalent graphical representation of Fig 4.9. The original model is depicted on the top of the figure and the transformed model at the bottom, below the arrow.

Alloy Analyzer provides instance enumeration (i.e. all possible instances that conform to the model transformation specification). Using this functionality we produced another random instance of the transformation. This instance is depicted in Fig. 4.10. We could continue to ask the Analyser to produce more instances of the transformation, however we are interested to check if the models, produced by the transformation satisfy certain properties.

4.3.2 Assertion checking

Assertions can be used to verify that a target model will always be well-formed given the transformation rules. For example we used the following assertion to validate that all InputPins in the target model have at most one incoming

```
ActivityEdge =
{(ActivityEdge[0]),(ActivityEdge[1]),
(ActivityEdge[2]),(ActivityEdge[3]),(ActivityEdge[4])}

target = {(ActivityEdge[0] , JoinNode[0] ),
(ActivityEdge[1],InputPin[0] ),(ActivityEdge[2] , InputPin[1] ),
(ActivityEdge[3] , FinalNode[0] ),(ActivityEdge[4] , JoinNode[0] )}

source = {(ActivityEdge[0] , InitialNode[1] ),
(ActivityEdge[1] ,InitialNode[0] ),(ActivityEdge[2] , JoinNode[0] ),
(ActivityEdge[3] , OutputPin[1] ),(ActivityEdge[4] , OutputPin[0] )}

ActivityNode =
{(OutputPin[0]),(OutputPin[1]),(JoinNode[0]),
 (InputPin[0]),(InputPin[1]),(InitialNode[0]),
 (InitialNode[1]),(FinalNode[0]),
 (CallAction[0]),(CallAction[1])}

outgoing = {(OutputPin[0] , ActivityEdge[4] ),
    (OutputPin[1] , ActivityEdge[3] ),
    (JoinNode[0] , ActivityEdge[2] )
Action = {(CallAction[0]),(CallAction[1])}
.
.
Mapping = {(Mapping[0])}

in2tin = { (Mapping[0] ,InitialNode[0] ,TInitialNode[0]),
    (Mapping[0],InitialNode[1], TInitialNode[1] )}

ae2tae = {(Mapping[0],ActivityEdge[0],TActivityEdge[0] ),
(Mapping[0], ActivityEdge[1] , TActivityEdge[3] ), (Mapping[0]
,ActivityEdge[3] , TActivityEdge[1] ),(Mapping[0] , ActivityEdge[4]
, TActivityEdge[2] )}

ca2tca = {(Mapping[0] , CallAction[0] , TCallAction[1] ),
(Mapping[0], CallAction[1] , TCallAction[0] )}


TInitialNode = {(TInitialNode[0]),(TInitialNode[1])}
TFinalNode = {(TFinalNode[0])}
TCallAction = {(TCallAction[0]),(TCallAction[1])} }
TActivityEdge = {(TActivityEdge[0]),(TActivityEdge[1]),
    (TActivityEdge[2]),(TActivityEdge[3])}
ttarget = {(TActivityEdge[0] , TInputPin[0] ),
    (TActivityEdge[1] , TInputPin[1] ),
    (TActivityEdge[2] , TInputPin[0] ),
    (TActivityEdge[3] , TFinalNode[0] )}
```

Fig. 4.8: Tuples representation of a random instance of the transformation produced by Alloy Analyzer

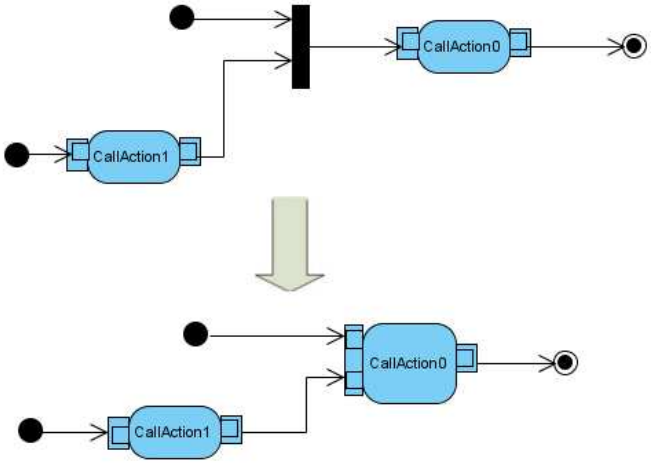

Fig. 4.9: Random Instance of the transformation produced by Alloy Analyzer

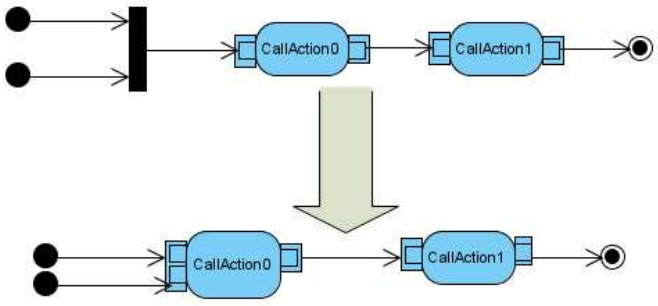

Fig. 4.10: Another random Instance of the transformation produced by Alloy Analyzer

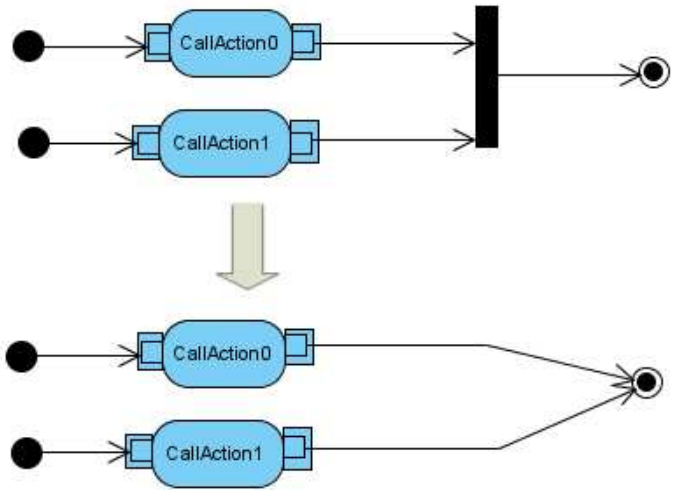

Fig. 4.11: Instance which violates the well-formedness rules of the target

edge:

```
all tip:TInputPin | #tip.tincoming < 2
```

Alloy Analyzer did not produce a counterexample for this assertion.

Additionally the well-formedness rules of the language require that a FinalNode has only one incoming edge [18]. 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. 4.11. The top of the figure depicts the instance of the source model and the bottom the instance of the target model that violates the well-formedness rules. [18], using their methodology have also discovered the same instance of the transformation that can produce a not well-formed target model. The malformed target model is produced when the *JoinNode* is connected to the *FinalNode*. When the *JoinNode* is removed, the edges that finished at the *JoinNode*, now finish at the *FinalNode*.

Alloy's ability to enumerate instances applies to assertions as well. As a result we can view all possible model transformations that violate our assertion. Alloy produces another counterexample, depicted in Fig. 4.12. It can be easily observed that when the JoinNode is removed, both *ActivityEdges* are connected to the FinalNode and thus produce a syntactically incorrect target model.

In order to resolve such problems, the transformation rules need to be augmented so that such a target model is not created.

4.4 Analysis on the Model Level

Alloy Analyzer's ability to simulate Alloy models renders it a potential model transformation execution environment. In this section we show how this can be achieved.

In Sect. 4.2 we showed how Alloy can be used to analyse model transformations on the metamodel level (i.e. by just defining the metamodels of the source and target language as well as the transformation rules in Alloy). However Alloy can also be used to simulate model transformations.

This is achieved by constraining the instances of the source metamodel. More specifically, assume the source model depicted in Fig. 4.9. It consists of two *InitialNodes*, two *CallActions*, one *JoinNode*, 5 *ActivityEdges*, etc. We can define this model as an input model to our transformation. Figure 4.13 depicts how this is achieved.

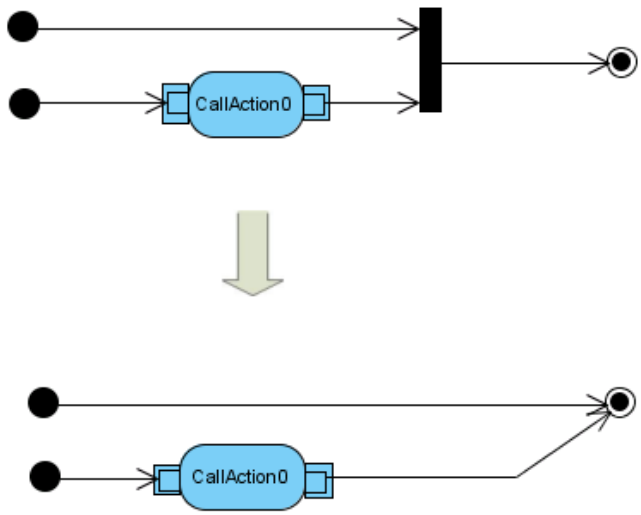

Fig. 4.12: Another instance which violates the well-formedness rules of the target

It can be observed that we define the model elements as singletons. For example, lines 1-2 define the two InitialNodes of the source model. The *one* keyword before the signature definitions, specify that the set will have only one element. Lines 3-4 define the two CallAction nodes, 5-9 the ActivityEdges and so on.

Lines 14 - 18 constraint that the only instances of the elements of the metamodel are those we have explicitly defined. For example line 14 constraints that the InitialNode set contains only the two InitialNode singleton sets that were specified in lines 1 and 2. Similarly line 15 constraints that there are two CallActions in the model, those that were defined in lines 3-4.

Finally lines 19 - 32 depict the structure of the source model. Lines 19 and 20 specify that *AE1* connects the *InitialNode1* to the *IP1*. Line 21 specifies that *IP1* belongs to *CA1*. Similarly the rest of the lines specify the associations between the model elements. Adding the representation depicted in Fig. 4.13 to the Alloy model, which represents the metamodels as well as the transformation rules, Alloy generates an instance of the target model.

We have therefore used Alloy Analyzer to simulate the model transformation of Fig. 4.9, by explicitly specifying the source model of the transformation.

```
//*****************************
// INSTANCE DEFINITIONS

1 one sig InitialNode1 extends InitialNode{}
2 one sig InitialNode2 extends InitialNode{}

3 one sig CA1 extends CallAction{}
4 one sig CA2 extends CallAction{}

5 one sig AE1 extends ActivityEdge{}
6 one sig AE2 extends ActivityEdge{}
7 one sig AE3 extends ActivityEdge{}
8 one sig AE4 extends ActivityEdge{}
9 one sig AE5 extends ActivityEdge{}

10 one sig Join1 extends JoinNode{}
11 one sig Final1 extends FinalNode{}
12 one sig OP1 extends OutputPin{} one sig OP2 extends OutputPin{}
13 one sig IP1 extends InputPin{} one sig IP2 extends InputPin{}

// ******* INSTANCE FACTS
fact{
14 InitialNode = InitialNode1 + InitialNode2
15 CallAction = CA1 + CA2
16 ActivityEdge = AE1 + AE2 + AE3 + AE4 + AE5 OutputPin = OP1 + OP2
17 InputPin = IP1 + IP2
18 JoinNode = Join1 FinalNode = Final1
}

//SPECIFY THE STRUCTURE
fact{
19 InitialNode1.outgoing = AE1
20 AE1.target = IP1
21 IP1.pac = CA1
22 CA1.pins = IP1 + OP1
23 OP1.outgoing = AE2
24 AE2.target = Join1

25 Join1.outgoing = AE3
26 AE3.target = IP2
27 IP2.pac= CA2
28 CA2.pins = IP2 + OP2
29 OP2.outgoing = AE4
30 AE4.target = Final1
31 InitialNode2.outgoing = AE5
32 AE5.target = Join1
}
```

Fig. 4.13: Definition of the instance of the source model in Alloy

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. [8] deduce metamodel coverage criteria and automatically produce test models. Küster et al. [18] suggest the use of a white box approach and propose three techniques for developing test cases. Baudry et al. [6] 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 [9]. Unlike their approach, our method is based on metamodelling and as a result it is not restricted to model refactorings.

Another approach for verifying model transformations comes from the domain of graph transformations. A model transformation can be represented as a graph transformation. More specifically Baresi et al. [5] present a case study where they use graph transformations theories to verify that their transformation is semantic preserving.

6. CONCLUSIONS

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 report 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.

BIBLIOGRAPHY

[1] UML2Alloy website. http://www.cs.bham.ac.uk/$\sim$bxb/UML2Alloy/index.php.

[2] David H. Akehurst and Stuart J. H. Kent. A Relational Approach to Defining Transformations in a Metamodel. In Jean-Marc Jezequel and Heinrich Hussmann, editors, *2002 - The Unified Modeling Language: Model Engineeing, Concepts, and Tools*, volume 2460 of *Lecture notes in computer science*. Springer, October 2002. ISBN 3-540-44254-5. URL http://www.cs.kent.ac.uk/pubs/2002/1559.

[3] David H. Akehurst, Behzad Bordbar, M. J. Evans, W. G. J. Howells, and Klaus D. McDonald-Maier. SiTra: Simple transformations in java. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006*, volume 4199 of *Lecture Notes in Computer Science*, pages 351–364, Genova, Italy, 2006. Springer.

[4] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, 2007. To Appear.

[5] Luciano Baresi, Karsten Ehrig, and Reiko Heckel. Verification of model transformations: A case study with BPEL. In *Second Symposium on Trustworthy Global Computing, TGC'06*, 2006.

[6] Benoit Baudry, Trung Dinh-Trong, Jean-Marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model transformation testing challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing*, 2006. URL http://www.irisa.fr/triskell/publis/2006/baudry06b.pdf.

[7] Jean Bèzivin, Fabian Büttner, Martin Gogolla, Frèdèric Jouault, Ivan Kurtev, and Arne Lindow. Model transformations? Transformation models! In *9th International Conference in Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *LNCS*, pages 440–453, Genova, Italy, 2006. Springer. URL http://wwwhome.cs.utwente.nl/$\sim$kurtev/files/Models2006.pdf.

[8] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *First International Workshop on Model, Design and Validation*, pages 29– 40, 2004.

[9] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[10] http://www.omg.org. MOF QVT final adopted specification. Document Id:ptc/2005-11-01. http://www.omg.org.

[11] IBM. Websphere Business Modeller. http://www-306.ibm.com/software/integration/wbimodeler/.

[12] Daniel Jackson. Alloy Analyzer website. http://alloy.mit.edu/.

[13] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, London, England, 2006.

[14] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer, 2006. URL http://www.lina.sciences.univ-nantes.fr/Publications/2006/JK06a.

[15] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2003. ISBN 032119442X.

[16] Jochen M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3): 233–259, 2006. doi: 10.1007/s10270-006-0018-8. URL `http://www.scopus.com/scopus/record/display.url?view=extended\&origin=resultslist\&eid=2-s2.0-33748305751`.

[17] Jochen M. Küster, Reiko Heckel, and Gregor Engels. Defining and validating transformations of UML models. In *HCC '03: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 145–152, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7803-8225-0.

[18] Jochen Malte Küster and Mohamed Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In Thomas Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 2006. ISBN 978-3-540-69488-5.

[19] T. Massoni, R. Gheyi, and P. Borba. Formal Refactoring for UML Class Diagrams. In *19th Brazilian Symposium on Software Engineering (SBES)*, pages 152–167, Uberlandia, Brazil, 2005.

[20] OMG. MOF Core v. 2.0, . Document Id: formal/06-01-01. `http://www.omg.org`.

[21] OMG. OCL Version 2.0, . Document id: formal/06-05-01. `http://www.omg.org`.

[22] OMG. UML Infrastructure, . Document: formal/05-07-05. `http://www.omg.org`.

[23] OMG. UML: Superstructure. Version 2.0, . URL `\url{http://www.omg.org/cgi-bin/doc?formal/05-07-04}`. Document id: formal/05-07-04. `http://www.omg.org`.

[24] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: Foundations*, volume 1. World Scientific, London, UK, 1997.

[25] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada*, pages 94–105. IEEE Computer Society, 2003.

[26] Dániel Varró and András Pataricza. Automated formal verification of model transformations. In Jan Jürjens, Bernhard Rumpe, Robert France, and Eduardo B. Fernandez, editors, *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, pages 63–78. Technische Universität München, September 2003.