# Using Traceability for Reverse Instance Transformations with SiTra

Seyyed Shah, Kyriakos Anastasakis, Behzad Bordbar
School of Computer Science
University of Birmingham, Edgbaston, B15 2TT. UK.
Email: {szs|kxa|bxb}@cs.bham.ac.uk

*Abstract*—**Model Driven Development and the core concept of Model Transformation has gained wide acceptance especially when used with UML languages. Model Transformations are used to map models in one language to another and can be used to transform a design model into an implementation or for analysing a design model to identify faults. However, transformations are a one time bridge and the instances of transformed models can not be automatically mapped back into the original language. This is despite the fact that a mapping must already exist between the two models. This mapping is represented in the *trace* of the transformations execution. Tracing is a feature of several transformation frameworks where by the source of every destination element is recorded.**

**Tracing has originally been applied for *change propagation* in chains of transformation and in debugging Model Transformations. In the current paper we present a novel use of Model Transformation tracing: for reverse instance transformation. That is, the automatic transformation of instances of the destination model back into instances of the source model. We demonstrate the method in a case study of UML2Alloy, a complex transformation from UML to the Alloy analysis language. In this case study, A UML Class Diagram is transformed in to its equivalent Alloy form. The presented method *automatically* transforms analysis (instances) of the Alloy model, back into UML-Object Diagram form that are valid instances of the original UML Class Diagram.**

## I. Introduction

Model Driven Development (MDD) [34] aims to promote the role of modeling in software development. Models in the context of MDD are captured in machine-readable representations, using languages which are widely adopted by the software industry [35]. Hence, it is possible to communicate such models to various parties and reuse them. This results in lower software production cost and shorter development cycles. Central to the MDD is the idea of automated transformation of models via tools, commonly known as Model Transformation Frameworks, that execute transformation automatically [4], [23], [1], [31]. A typical Model Transformation framework accepts three inputs, a metamodel of the source language, a metamodel of destination language and a specification of the transformation which maps the model elements of the source to the destination. Then, for any given model complying to the metamodel of the source the tool executes the transformation resulting in the creation of an instance of the metamodel of the destination.

In complex applications domains, MDD can also be used to create multiple models of the systems automatically to bridge the gap between technical spaces [22]. For example, MDD can be used to create analysable models from a design model [21], [3]. In such cases, the result of the analysis must also be transformed back to be presented to the designer. In this paper we demonstrate that traceability is an important issue in such Model Transformations. In effect traceability is a mechanism for recording the link between the source and target model elements [11], [29]. Establishing such links allows defining the reverse transformations automatically.

This paper reports on our current research on extending the Simple Transformer (SiTra) [1] with traceability capabilities. SiTra is a simple and lightweight implementation of an extensible Transformation Engine. SiTra has been successfully applied to model transformation in various application domains[2], [8], [7]. The paper also reports on a case study involving application of new version of SiTra to a complex Model Transformation from UML to Alloy [7].

The paper is structured as follows, the next section gives a brief background to the subject at hand. The concepts of Model Driven Development and Model Transformation, tracing transformations and the transformation framework: SiTra. Next we we shall describe the problem followed by an outline of the solution in terms of the architecture and algorithm, as well as how SiTra was modified to accommodate traceability. Finally we present an example Model Transformation in UML2Alloy that applies the solution to a specific problem, followed by some discussion.

## II. Background

### A. Model Driven Engineering

Model Driven Architecture (MDA) [25], a flavour of MDD which is initiated by the Object Management Group (OMG). MDA makes use of Meta Object Facility (MOF) [28] which describes the metamodels. Metamodels are themselves *models*, from which models of the system are instantiated. MOF can be compared to EBNF, which is used for defining programming languages grammars. As a result, MOF is a blueprint from which MOF Compliant metamodels are created.

Figure 1 depicts an outline of MDA and the process of Model Transformation. A number of Transformation Rules are used to define how various elements of one metamodel (source metamodel) are mapped into the elements of another metamodel (destination metamodel). The process of Model Transformation is carried out automatically via software tools

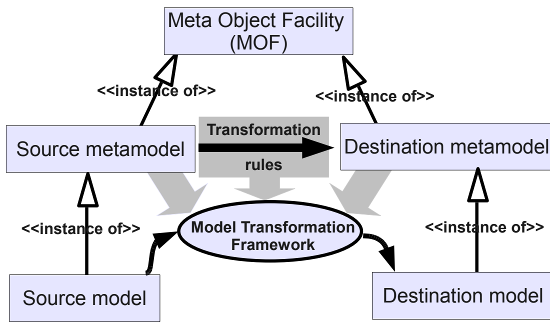



Fig. 1. An Overview of MDD

which are commonly referred to as Model Transformation Frameworks [37], [1], [13]. A typical Model Transformation Framework requires three inputs: source metamodel, destination metamodel and Transformation Rules. For any instance of the source metamodel, a *Transformation Engine* executes the rules to create an instance of the destination metamodel.

The central concept of MDD is Model Transformation [32], the mechanism for bridging technical spaces. Model transformations take as input one or more source models that conform to source language and translate them into one or more target models in the context of a destination language. Judson et al. [20] propose that transformations can be applied from one of two dimensions; vertical, that change the context for example Design to code or horizontal, as analysis for example UML2Alloy.

### *B. Model Transformation Traceability*

Traceability is a feature in a Model Transformation Engines that keeps record of which element(s) in the source model maps to which element in the destination. Bondé et al. [5] apply the traceability to *change propagation*, thus if the source changes slightly, the change can be reflected in the destination without re-running the entire transformation. Change propagation is most useful when several successive transformations are applied to a model, so the models can be made interoperable. Moreover, the ability to trace the source of an element has been used in debugging of Model Transformation [16]. Thus, traceability support is a desirable part of a Transformation Engine feature set and for developer support.

Jouault [19] identifies two groups of model transformation traceability stratergies, automatic or manual tracing. Automatic tracing requires no manual intervention by the developer, the trace information is recorded transparently during the transformation. Manual tracing, as the name suggests, requires explicit tracing rules be defined as part of the Model Transormation to be traced. Each method has it's merits, automatic tracing requires little developer intervention and leads to less cluttered transformations; manual tractability gives the developer control over what information is traced. For further classification of tracing strategies in frameworks, see [11]. It is clear that intergrated traceability support in the form of automatic tracing is a desirable feature for a Model Transformation framework.

The requirements for traceability information vary between frameworks. Vanhooff and Berbers [36] encode traceability information into a *UML Profile for Traceability*, where a model of the transformation is extended with trace information. The approach is taken so that large Model Transformations can become smaller transformations as part of "Transformation Chains". The smaller transformations are more axiomatic so have dependencies on preceding transformations outcomes and rely on trace information. The Kermeta [12] framework with a similar aim of enabling chains of transformations, defines a distinct meta-model of traceability information. So a model of the trace is populated at transformation time. The OMG's QVT [29] Model Transformation specification aims for a generic approach to transformation traceability, defining the *Trace Class* and *Trace Instance* entities. Trace instaces are created with the appropriate information during transformation. The precise form of traceability support across languages depends on the motivation for adding traceability to a framework.

### *C. **Si**mple **Tra**nsformer (SiTra)*

There are a wide range of languages available to specify MDD Transformation Rules [4], [13], [29]. Such languages, which are mostly extending [30], not only provide strong constructs for the specification of Model Transformations, but also are supported by Model Transformation Frameworks for executing the transformations. However, none of these languages are widely adopted by the academic community or industrial tool vendors. Much anticipated Query View Transformation (QVT) by OMG is now finalised [29] and is expected to result in a unifying language for specifying transformations. To execute a specification of a Model Transformation in the above languages, they must be transformed into lower-level languages such as Java.

In a large project, it is possible to divide the specification and implementation of Model Transformations between two different groups of people who have relevant skills. In the case of smaller groups of developers and newcomers to MDD, the combined effort involved in becoming an expert in the two sets of skills described above is overwhelming. In particular, the steep learning curve [24] associated with current MDD tools, such as [10], [9], [14], is an inhibitive factor in the adoption of MDD by even very experienced programmers. Simple Transformer (SiTra) [1] is a simple and lightweight Model Transformation Framework aiming to use Java for both writing Model Transformations and providing a minimal environment (the execution engine) for transformation execution. SiTra consists of two interfaces, the *Rule* interface, which user defined mapping rules have to implement and the *Transformer* interface, which provides the skeleton of the methods that carry out the transformation. The authors of SiTra provide a simple implementation of the *Transformer* interface and to use SiTra for simple transformations. The modeller only needs to define the transformation rules by implementing the *Rule* interface, which consists of three methods: *check()*, *build()* and *setProperties()*. If the rule is applicable for the *source* element in question, the *check()* method of the rule interface returns

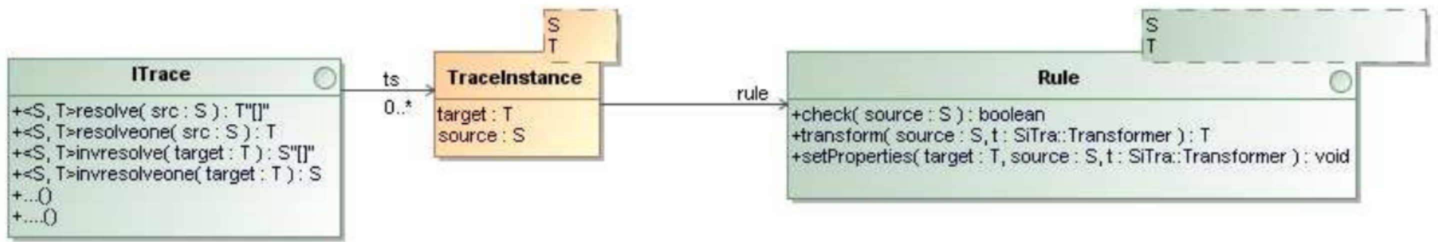



Fig. 2. A Model of the Tracing Mechanism

true and the *build()* method is executed. The *build()* method generated the target model element. The *setProperties()* is used to set the attributes and links of the newly created target element. SiTra has been successfully applied to Model Transformation in various application domains[2], [8], [7]. For further details on SiTra please refer to [1].

## III. Outline of the Approach

### A. Description of the Problem

Often a Model Transformation is used to produce a model of a target language as an intermediate step in a process. For example, a model transformation may be used to transform a model from a source language *A* to a target language *X*, to take advantage of more advanced tool support in *X*. In such a case, the toolset of the target language is used to process (e.g. analysis, refactoring) the produced model. The results of this processing need to be interpreted in the domain of the original language *A*. If the transformation between the source and the target languages is not bidirectional this is not a simple task. In this section we present an algorithm, which uses the tracing information of a Model Transformation to interpret the results of a process on the target model, using concepts of the source language, in a unidirectional transformation. Moreover we propose extending the SiTra framework with support for traceability, to implement our method.

### B. Architecture and Algorithm

Figure 3 depicts an overview of the method proposed. On the metalevel (*M2*) elements of the *Source* and *Target* metamodels (*A* and *X* respectively) are mapped using the Model Transformation *T*. On the model level (*M1*) if a source model *B* is given as input to the transformation a target model *Y* will be automatically generated. The execution of the Model Transformation will also generate a set of traces *T'*, which record how each element of the source model is mapped to an element of the target model. If *Z* is an instance of the target model on the *M0* level, using the trace information *T'*, we can construct *C*, which is an instance of the source model *B*.

The algorithm to extrapolate C is as follows. For all elements in *Z*, find the source of the parent element from *T'* and create an instance of that element. Once all elements in *Z* are applied to the algorithm, the resulting instance model *C* is produced. This is a valid instance model of B at M0-level in the source language, created automatically. The algorithm is shown in Figure 4.

### C. Tracing Support in SiTra

In this section we describe how the SiTra Model Transformation engine is modified to add traceability support. SiTra, was developed by Akehurst et al. [1] as a simple Object Oriented Transformation Engine, in principle based on a modified visitor pattern. The engine has been modified to support both Model to Model (*M2M*) and Model to Text (*M2Text*) transformation tracing.

The MOF Queries Views and Transformations (QVT) specification [29] defines a tracing mechanism that can be used to trace which source metamodel elements are mapped to which target metamodel elements and the inverse. Based on the tracing mechanism of the QVT specification, we have developed and implemented an extension to the SiTra Transformation Engine. Figure 2 depicts our model for tracing *M2M* transformations.

More precisely, our tracing consists of an interface (*ITrace*), which holds a collection of *TraceInstances* (*ts*). Each TraceInstance, represents a mapping between a source and a target model element, through a SiTra rule. An implementation of the *ITrace* interface, provides a number of methods to query the *ts* collection. More specifically the *resolve* method, queries the *ts* collection, and returns all target instances that have been created during the transformation, from the *src* instance. Likewise, *resolveone* should return only the first instance of the target element that has been created during the transformation from the *src* instance. The method names preceded with 'inv' (i.e. *invresolve*), perform the inverse (i.e. return the source elements that have been mapped to the target element passed as a parameter). The QVT specification, also defines a number of additional methods that can be used to query the trace model.

To populate this tracing model, we have extended the implementation of the *transform* method of the SiTra distribution. Our implementation ensures that, for each rule being executed, a trace is recorded (i.e. a *TraceInstance*), which keeps track of the instance of the SiTra rule responsible for the transformation, the instance of the source metamodel provided as input to the rule and the instance of the target metamodel produced by the rule.

A similar tracing model has been implemented in SiTra for Model to Text (*M2Text*) transformations, which are usually defined in order to transform the abstract syntax model elements to a textual notation. Each *TraceInstance* of the M2Text trace model maps an element of the metamodel of the languages,

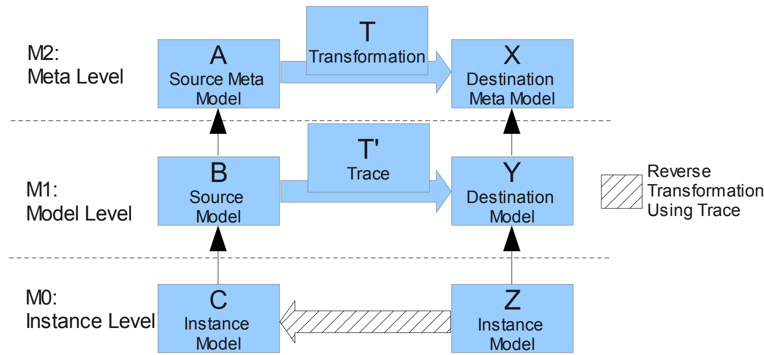



Fig. 3. Architecture of the Proposed Solution

**Input**: Instance Model: Z, Trace: T', Model: Y, Model: B
**Output**: Instance Model: C
**foreach** *element z in model Z* **do**
  find the class element of z, y from the model Y
  query T' using y to find class element b, the source of y
  using z, create c in instance model C
**end**

Fig. 4. Algorithm for Reverse Instance Transformation

to a range in the generated text model. The range is identified by the rows and columns it occupies in a text file.

## IV. Case Study:UML2Alloy

To demonstrate the method, we apply the approach to a model transformation from UML class diagrams enriched with OCL constraints [35], to the Alloy language [18]. Alloy is an increasing popular textual language based on first-order logic and fully automated analysis capabilities. The transformation from UML to Alloy is implemented as part of a tool called UML2Alloy [3], which uses the SiTra transformation framework. UML2Alloy has successfully been applied to the analysis of agile manufacturing [7], E-Business applications [6] and Security in E-Commerce systems [15].

UML is now widely accepted for designing systems before implementation, allowing better modelling as a part of the development process. The idea behind UML2Alloy is to allow the designer to specify the system using UML and simultaneously harness the analysis capabilities of the Alloy language to identify possible faults in the created design via the Alloy language. More precisely, the Alloy language is supported by a tool called Alloy Analyzer, which is able to automatically simulate an Alloy model by creating a arbitrary instance of the model. Additionally the analyser offers the ability to debug overconstrained models [33], by locating the statements which are responsible for the inconsistent model.

Figure 5 depicts an overview of the problem addressed in this paper in the context of UML2Alloy. The UML2Alloy transformation maps elements of the UML and OCL metamodel to elements of the Alloy metamodel, which has been developed using the Alloy grammar [3]. If a UML model is given to the UML2Alloy implementation, it can automatically create an Alloy model. This Alloy model can then be automatically analysed, by exploiting the Alloy Analyzer API. If the model is consistent the Alloy Analyzer will create a arbitrary instance of the model in XML form. If the model is overconstrained, however, it will return the lines and columns of the statements responsible for the inconsistency.

In both in simulation and analysis we need to represent the results of the outcome of the Alloy Analyzer, from the Alloy language concepts back to the UML domain. If the analyser provides an instance of the model, we need to represent the instance in terms of a UML Object Diagram [35]. If, on the other hand, the analyser returns the regions in the Alloy text file that are responsible for an overconstrained Alloy model, we need to locate the original UML model elements responsible for the inconsistency and present them to the user. Since the transformation from UML to Alloy is not bidirectional [34] (e.g. the multiplicity constraints of a UML association are mapped to an Alloy *fact*, but an Alloy *fact* does not necessarily correspond to UML association multiplicity constraints), we need to employ the technique presented in the previous section to carry out reverse instance transformation. How this technique is used in UML2Alloy, is described in the next section with the help of an example.

In order to take advantage of the analysis capability of the Alloy language, for the designer who wishes to use *only* UML, we will employ the tracing techniques presented in the previous section. Figure 6 shows the proposed solution, in terms of Alloy. Given a UML2Alloy transformation and trace, it is possible to transfer the outcome of the analysis into the UML form. In the case where the model is simulated to generate instances, we transform the instance back into UML Object Diagram, following the algorithm shown in

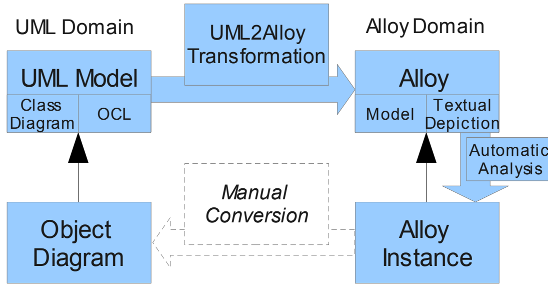



Fig. 5. UML2Alloy Case Study: Problem of Reverse Instance Transformation

Figure 4. In this case, for every instance element in the Alloy Instance model, find the signature in the Alloy Model. From the Trace, find the source Class in the UML model of the Signature. Finally instantiate the Class using the data from the Alloy instance in a UML Object Diagram. If repeated for all elements in the Alloy instance, an Object Diagram representation of the Alloy instance is created. When there is an inconsistency found in the model by Alloy Analyzer, the position in the text model is given. This position can be traced back to the cause in the UML, using the same algorithm.

In the next section contains a brief introduction to the Alloy language. Following this, there is an example model and model transformation in UML2Alloy where the outcome of the analysis are transfered back into UML form.

### A. The Alloy Language

In this section we present a brief overview of the Alloy language [18] and supporting tool, the Alloy Analyzer. The Alloy language was designed for automatic reasoning and analyse of software systems. The five high-level constructs (termed paragraphs) in the language are Signatures, Facts, Predicates, Functions and Assertions. Multiples of these are used in the construction and analyse of a model. There is also the Run sentence, required to initiate analysis of a model. For the definition of models, Signatures, Functions and Facts in analogy to UML are classes, methods and constraints respectively. For analysis there are Predicates, used to simulate the models' properties with sample instances. Assertions are used in analysis in attempt to (in)validate a particular property of a model by producing counter-examples. The notion of model scope is important as Alloy's underlying logic is First-Order. As first order logic is undecidable in the general case, a scope must be set at execution time to bound the analysis space.

Using the Alloy Analyzer, a model in the Alloy language automatically be analysed and simulated. Models in the Alloy languages are translated into a series of boolean expressions. This form is suitable for analysis via off-the-shelf satisfiability (SAT) solvers. Simulation of the Alloy model is used to generate valid instance models.

The process is that the Alloy Analyzer automatically translates an Alloy model to a SAT formula that can be analysed by SAT solvers. In the case of assertion, the statement is negated and the Alloy Analyzer tries to find an instance of the model

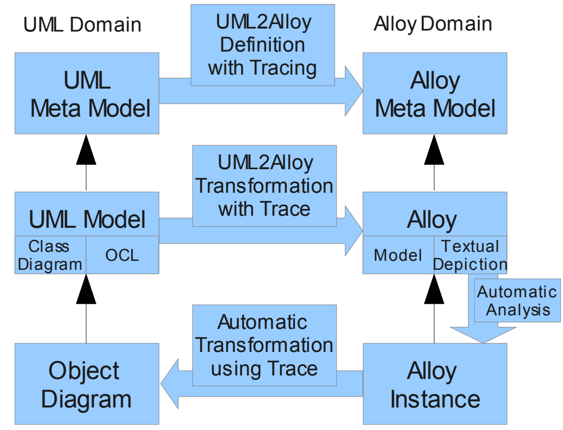



Fig. 6. UML2Alloy Case Study: Proposed Solution Using Tracing

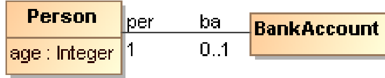
that conforms to the negated statement. If it can find one, this is a counterexample, if it cannot find one, the assertion may be valid. In the case of simulation, it searches for cases for an expression evaluates which to true, this expession will encode an arbitrary instance of the model.

## V. Example

In this section we introduce the example model and transformation to demonstrate the tracing mechanism. The UML model (Figure 7) is transformed into the Alloy model (Figure 8) using the UML2Alloy trasformation in SiTra. The first part of each model (Figures 7a, 8a) is consistent in either language; conforming instances can be created by hand in UML or automatically in Alloy. We use the first models (Figures 7a, 8a) to demonstrate reverse instance transformation. The second part of each model (Figures 7b, 8b) extends on the first valid part to add contradictory constaints, making the original model inconsistent. The inconsistent model has no valid instances in either language and this property of the models is used to demonstrate the reverse instance tracing, to discover the source of the inconsistency in the original UML. The textual Alloy models presented here are refactored slightly to aid readability and for purposes of brevity.

We use a two-class, UML model shown in Figure 7 as the example in this section. According to the model, one *Person* can be associated either one or no *Bank Account*s. The *Person* class has a single integer attribute representing age. The first part of the model (Figure 7a) is legal UML, valid instance of this model can be created by hand, as UML Object Diagram. In the second (Figure 7b), invalidating part of the model, two OCL constraints are added to the model specifying purposely contradictory constaints about the age of a *Person*. Both parts of the models are transformed into the Alloy model, shown in Figure 8.

The Alloy model (Figure 8a) consists of two signatures (*sig*), *Person* and *Bank Account*, here the keyword *some* enforces the existence in an instance model. *Person* has the two atoms, *age* is a relation to one integer and *ba* is a relation one or no (*lone*) *Bank Account*. The first fact paragraph *fact*{




(a) Valid Elements of Model

```
context Person inv first:self.age>20
context BankAccount inv second:self.per.age<18
```

(b) OCL Constraints (Introducing Contradiction)

Fig. 7. Example UML Model

*per* = ∼*ba* }, signifies that *ba* and *age* form a single and the same relation from *Bank Account* to *Person* and vice-versa. Simulation of the first part of the model should yield sample instances using the Alloy Analyzer. In the second part of the Alloy model (Figure 8b) two facts constrain the model in contradictory way; that is the age of a Person must be both less then eighteen and greater then twenty.

In this example, mapping between the two models is performed by the UML2Alloy tool automatically, we explain the mapping here for the sake of clarity. The UML classes *Person* and *Bank Account* are transformed into signatures of the same name. The *age* attribute in person becomes the integer field *age* in the *Person* signature. The association is transformed into fields in the respective signatures, named *per* and *ba* and facts representing multiplicity. The OCL constraints are transformed into facts in the Alloy model. In UML2Alloy the above is a two step process, the first transformation is to a MOF like model of Alloy which is mapped via a one-to-one transformation into the Alloy textual form.

In the following two sections, we use the example to demonstrate the traceability from two perspectives. Firstly, given valid models (Figures 7a, 8a), transform the Alloy instances back into UML form. Secondly, given an invalid UML model (Figure 7), trace back the element(s) causing the inconsistency in the UML, as discovered in the Alloy model (Figure 8a) Alloy Analyzer.

### A. Example One: Reverse Instance Transformation

The Figure 9, shows the output of analysis from the Alloy Analyser in XML form, our method will automatically create an Object Diagram form of this instance. It Depicts a sample of the output Analysis of the Alloy Analyser. In this partial sample, there is an instance (atom) of the *Person* signature from the Alloy model, labelled as *Person$0*. There are also two fields, that form tuples (relations) from *Person$0* to an integer (5) for the *age* atom. The second*BankAccount$2* via the *ba* atom, instances of the*Person* signature's atoms.

Reverse instanstance transformation, this context, is taking an Alloy instance model and transforming it to back into the original UML form. Although this may seem a straight forward problem, possibly solved by transformation, often times there is a difference in semantic expressiveness of elements between the source and destination language. Reverse instance transformation is not possible in all cases using only the source and destination model as the transformation can change from a specific to a general one.

```
some sig Person{
age : one Int,
ba : lone BankAccount}

some sig BankAccount{
per : one Person}

fact{ per = ~ba }
fact{per in BankAccount lone->one Person}
fact{ba in Person one->lone BankAccount}
```

(a) Alloy Model, Valid Segment

```
fact{all p : Person | int p.age > 20}
fact{all b : BankAccount | int b.per.age < 18}
```

(b) Alloy Model, Contradictory Segement

Fig. 8. Alloy Model Resulting from Transformation of Figure 7, using UML2Alloy

We demonstrate the issue in the example in Figure 9 where it is not possible to know the origin (in the UML model) of the Alloy instance atom marked *age*. This is because an Alloy field could have been mapped from a UML attribute or a UML association. The issue becomes more apparent in the instance (Figure 9) where the original association between *Person* and *Bank Account* has become a pair of Alloy atoms *ba* and *per*. In Alloy, these atoms are semantically equivalent to the *age* atom and thus indistinguishable for instance creation, without trace information.

For this example we assume the Model transformation of the consistent UML model in Figure 7a into the consistent Alloy model Figure 8a. We shall create a UML representation of the instance (Figure 9), which was created by analysis of the Alloy model (Figure 8a). The method of finding the origin in UML using the algorithm in the case of Figure 4 is as follows. Take the instance tuple *age* that maps the atom *Person$0* (Figure 9)to the *Int* 5, find the parent element in the Alloy model (Figure 8). The parent element is the atom *age*, in the signature *Person*. The origin of this element is found by querying the trace, the origin here is is the attribute *age* of *person*, in the source UML model (Figure 7). At this point, it is possible to instantiate the age attribute in the UML Object Model of *Person$0*, using data from the Alloy tuple i.e. with a value of 5.

### B. Example Two: Model Inconsistency Tracing

Due to design errors it may be possible to define an inconsistent model, such as the model shown in Figure 8b. Discovering the cause of design inconsistencies in a model is highly desirable. A UML model can be transformed into Alloy and simulated to find inconsistency using the Alloy Analyzer. An inconsistent model in Alloy will result in no instance being generated. However the offending sentence(s) will be know from the analysis in the Alloy model, but not the original UML. The Alloy Analyser uses UnSat Core [17] feature of the SAT solver to find the line and column numbers of the inconsistent portion. This analysis does not

```
...
<sig name="Person" extends="univ">
  <atom name="Person$0"/>
</sig>

<field name="age">
    <type> <sig name="Person"/> <sig name="Int"/> </type>
    <tuple> <atom name="Person$0"/> <atom name="5"/> </tuple>
</field>

<field name="ba">
    <type> <sig name="Person"/> <sig name="BankAccount"/> </type>
    <tuple> <atom name="Person$0"/> <atom name="BankAccount$2"/> </tuple>
</field>
...
```

Fig. 9. Example One: Partial XML Output from Alloy Analyzer

apply directly to the UML model, however our method is able to trace the source of an element in the original UML and thus the originating cause of inconsistency. UML2Alloy allows inconsistency to be uncovered via analysis in Alloy and our method uncovers the root cause of the inconsistency in the original model, using the trace.

Our solution will trace the root cause of inconsistency in a UML model, after it has been transformed and analysed using Alloy. The method is as follows, assume the Model Transformation from the inconsistent UML in Figure 7 (combined) to the Alloy representation of the same in 8. The Alloy Analyser will uncover the inconsistency and the place (line(s), column(s)) at which the problem occurs in the Alloy model. To find the cause of the inconsistency in UML, first identify the element in the model causing the error. In this case, the UnSat Core gives the expressions from the two facts in 8b as the cause. To find the error in the UML Model, query the trace using the given ranges returns the result in OCL form, shown in Figure 10 and the location in the source model. This will highlights the two conflicting statements in the UML form for the developer to maneuver as necessary. The strength of the method is apparent in a large model with potentially many complex or similar constraints, as it is possible to pinpoint precisely which statement caused the inconsistency. We have shown that using our method to reverse transform elements in the destination model, it is possible to find the source of an inconsistency in the original UML model.

```
[self].age>20
[self].per.age<18
```

Fig. 10. Example Two: Result of Text to Model Tracing

## VI. Related Work

A common motivation for traceability in the literature has been to support chains of transformation, sequences of transformations with dependency, as found in [12], [36], [19]. The approach used in these methods is to add specific rules for traceability information and create a separate model of traceability as output of transformation.This model is then used as input for further transformation stages. So the tracing approach for chains of transformations with dependency is fundamentally different from ours. In [19], an important discussion is made about generating trace information from a generic perspective based on an approach similar to [36], [12]. Our approach generates trace information automatically which is used internally in the context of the original transformation. We have demonstrated that it is possible to generate useful trace information automatically, without requiring specific rules in the original transformation.

The extension of the SiTra implementation to support tracing of model transformations, has been inspired by the QVT standard [29]. The QVT standard provides a number of methods to query the transformation traces, while the transformation is being executed (i.e. provides the ability to access target objects created from source objects and the inverse). Since the QVT specification does not provide any guidelines on how to implement the standard, the SiTra extension presented in this paper, which follows the QVT naming convention for methods, can be considered as an implementation of the tracing mechanism of the QVT standard.

Additionally the SiTra extension to provide tracing for model to text transformations is based on another OMG standard, the MOF Models to Text Transformation language specification [27]. In particular, the metamodel of the standard provides the ability to trace from which model element, a block of text has been created. An implementation of model to text tracing is presented in [26]. Our metamodel for model to text tracing is similar to the metamodel presented in [26]. However, these methods are more general than our extension to SiTra, since they address issues like synchronisation and change propagation between a model and its textual representation, if either the model or the textual representation changes. Our model to text transformation is simpler, since the generated Alloy textual model, is used as an intermediate step in order to be able to exploit the Alloy Analyzer API.

## VII. Conclusion and Future Work

This paper presents a implementation *Traceability*, a mechanism for recording the link between the source and target

model elements in Model Transformation Frameworks. Traceability is often used for the management of Model Transformation process for change propagation or debugging purposes. In this paper, Traceability produces the reverse transformation automatically at the instance level. Such applications of Traceability are essential for horizontal Model Transformations such as UML2Alloy.

The presented approach, which is inspired by QVT, is formulated as an algorithm. The SiTra framework is extended to a new version to do both model-to-model and model-to-text transformations, while the algorithm is implemented to allow traceability in both cases. The paper also reports on a case study based on the model transformation in UML2Alloy, which uses the new version of SiTra. Using a brief example, the process of Model Transformation carried out via UML2Alloy is described and the role of tracing mechanism is explained.

In the current implementation, only the binary transformation rules are traceable. So to handle ternary rules, they must be modified to create multiple binary rules. Discovery of methods of extension to ternary rules remains a subject for future research. There is a clear scope for extending the current framework to fulfill the original objectives of Traceability such as change propagation and debugging. This also remains an area for future research.

## References


[1] David 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, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 351–364. Springer, 2006.

[2] DH Akehurst, O. Uzenkov, WG Howells, and KD McDonald-Maier. Compiling UML State Diagrams into VHDL: An Experiment in Using Model Driven Development. *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences), Genova, Italy*, 2007.

[3] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In G. Engels, B. Opdyke, D.C. Schmidt, and F. Weil, editors, *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 436–450, Nashville, USA, 2007. Springer.

[4] J. Bézivin, F. Jouault, and D. Touzet. An Introduction to the ATLAS Model Management Architecture. *Research Report LINA,(05-01)*, 2005.

[5] Lossan Bondé, Pierre Boulet, and Jean-Luc Dekeyser. Traceability and interoperability at different levels of abstraction in model transformations. In *Forum on Specification and Design Languages, FDL'05*, 2005.

[6] Behzad Bordbar and Kyriakos Anastasakis. MDA and Analysis of Web Applications. In *Trends in Enterprise Application Architecture (TEAA) 2005*, volume 3888 of *Lecture notes in Computer Science*, pages 44–55, Trondheim, Norway, 2005.

[7] Behzad Bordbar and Kyriakos Anastasakis. UML2Alloy: A tool for lightweight modelling of Discrete Event Systems. In Nuno Guimarães and Pedro Isaías, editors, *IADIS International Conference in Applied Computing 2005*, volume 1, pages 209–216, Algarve, Portugal, February 2005. IADIS Press.

[8] Behzad Bordbar, Gareth Howells, Michael Evans, and Athanasios Staikopoulos. Model transformation from owl-s to bpel via sitra. In *ECMDA-FA*, pages 43–58, 2007.

[9] Codagen Technologies Corp. Codagen architect.

[10] Borland Software Corporation. Borland together architect.

[11] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[12] J.R. Falleri, M. Huchard, and C. Nebut. Towards a traceability framework for model transformations in Kermeta. *ECMDA-TW Workshop*, 2006.

[13] F. Fleurey, Z. Drey, D. Vojtisek, and C. Faucher. Kermeta language. *Reference manual, Internet: http://www.kermeta.org/docs/kermeta-manual.pdf*, 2006.

[14] F. Flore. MDA: The Proof is in Automating Transformations Between Models, 2003.

[15] Geri Georg, Indrakshi Ray, Kyriakos Anastasakis, Behzad Bordbar, Manachai Toahchoodee, and Siv Hilde Houmb. An Aspect-Oriented Methodology for Developing Secure Applications. *Information and Software Technology. Special Issue on Model Based Development for Secure Information Systems*. Accepted for publication.

[16] Mark Hibberd, Michael Lawley, and Kerry Raymond. Forensic debugging of model transformations. In *MoDELS*, pages 589–604, 2007.

[17] D. Jackson, M.I.T.I. Shlyakhter, and M. Sridharan. alloy: a logical modelling language. *Proceedings of 3rd International Conference of B and Z Users (ZB 2003): Formal Specification and Development in Z and B, Turku, Finland*, page 1, 2003.

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

[19] F. Jouault. Loosely Coupled Traceability for ATL. *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, 2005.

[20] S.R. Judson, D.L. Carver, and R. France. A Metamodeling Approach to Model Refactoring. *Submitted to UML03*, 2003.

[21] Soon-Kyeong Kim. *A Metamodel-based Approach to Integrate Object-Oriented Graphical and Formal Specification Techniques*. PhD thesis, University of Queensland, Brisbane, Australia, 2002.

[22] I. Kurtev, J. Bézivin, and M. Aksit. Technological Spaces: an Initial Appraisal. *CoopIS, DOA*, 2002, 2002.

[23] M. Lawley and J. Steel. Practical Declarative Model Transformation With Tefkat. *MoDELS Satellite Events*, pages 139–150, 2005.

[24] G.A. Lewis and L. Wrage. Model Problems in Technologies for Interoperability: Model-Driven Architecture. Technical report, Carnegie Mellon University, Software Engineering Institute, 2005.

[25] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, OMG, 2003.

[26] Gøran K. Olsen and Jon Oldevik. Scenarios of traceability in model to text transformations. In *ECMDA-FA*, pages 144–156, 2007.

[27] OMG. *OMG: MOF Model to Text Transformation Language*. OMG, 2004.

[28] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*. OMG, 2004.

[29] OMG. *MOF QVT Final Adopted Specification*. Object Modeling Group, June 2005.

[30] Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. In *ER*, pages 449–464, 1998.

[31] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. The epsilon generation language. In *ECMDA-FA*, pages 1–16, 2008.

[32] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[33] 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.

[34] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[35] OMG UML. 2.0 superstructure final adopted specification. *OMG Document reference ptc/03-08*, 2, 2003.

[36] B. Vanhooff and Y. Berbers. Supporting Modular Transformation Units with Precise Transformation Traceability Metadata. *ECMDA-TW Workshop, SINTEF*, pages 15–27, 2005.

[37] M. Volter. OpenArchitectureWare 4. *Bericht, openArchitectureWare*, 2007.