Opening the Black-Box of Model Transformation

John T. Saxon[1](✉), Behzad Bordbar[1], and David H. Akehurst[2]

[1] University of Birmingham, Birmingham, UK
{j.t.saxon,b.bordbar}@cs.bham.ac.uk
[2] Itemis AG, 44536 Lünen, Germany
dr.david.h@akehurst.net

Abstract. The automated execution of model transformation plays a key role within Model Driven Development. The software that executes a transformation, commonly known as a transformation engine, receives the meta-models of the source and destination, and a set of transformation rules as input. Then the engine can be used to convert instances of the source meta-model to produce a destination model. Transformation engines are often seen as black boxes. In order to be sure of the correct execution, it is crucial to understand how a transformation engine executes a given transformation. This paper presents a method of capturing and analysing the activities carried out within the transformation engine by elaborating on existing tracing mechanisms used by existing engines. We compare the tracing mechanisms involved in four popular, rule-based transformation frameworks and highlight their shortcomings. A new trace meta-model is presented to deal with some of these shortcomings. These processes can be applied to all existing frameworks; as a proof of concept we have extended an existing traceability framework, based on our earlier work, to implement these mechanisms.

1 Introduction

The execution of model-to-model (M2M) transformations is often viewed as a black box process. Transformation engines such as the Epsilon Transformation Language (ETL) [16] and the ATLAS Transformation Language (ATL) [15] require the meta-models of the source, destination and a set of transformation rules as input. Then a transformation engine, behind the scenes, automatically executes the rules and converts a source model to generate the destination model. Even during testing and verification, all existing research focuses on correctness of rules, while treating the transformation engine as a black-box that is assumed to execute correctly. One exception to this "black-box" routine is the process of *tracing* [1,9,18]. Traceability can be supported in transformation engines and gives access to the linkage between source and destination models established by a transformation execution [18]. To the best of our knowledge the first tracing mechanism, within non-graph based transformation engines, was implemented and used by UML2Alloy [21] through the Simple Transformer (SiTra) [2]. UML2Alloy produces Alloy models from a UML class diagram and

© Springer International Publishing Switzerland 2015
G. Taentzer and F. Bordeleau (Eds.): ECMFA 2015, LNCS 9153, pp. 171–186, 2015.
DOI: 10.1007/978-3-319-21151-0_12

OCL statements via a transformation. In Shah et al. [21], the transformation trace was used to convert a counter example produced by Alloy back to UML.

This paper is based on our study of four model transformation frameworks: ATLAS Transformation Language (ATL) [15], Epsilon Transformation Language (ETL) [16], Operational Query/View/Transform (QVT-O) [18], and the Simple Transformer [2]. We have identified a number of shortcomings of the existing frameworks with respect to traceability mechanisms implemented within them. In this paper we focus on three issues: orphan objects, loss of information regarding the ordering of execution and the dependencies between the rules. These shortcomings and their adverse effects, which are common to most frameworks, are described with the help of well-known examples. Then we explain a modification to the design of transformation engines that can eliminate these deficiencies. We present an implementation of the design by extending SiTra. We also describe the changes required to modify ETL to compliment the design. This is to show that other engines can adopt our design easily. Finally we evaluate, the approach by mapping a relational database to Apache HBase [22], a NoSQL database, via a non-trivial transformation. This transformation is different from most transformations specified on the relational databases as both the data is migrated and the schemas are mapped. In particular we report on the execution of the transformation on the so-called employee database provided by MySQL. This dataset contains four million rows over six tables and has been successfully transformed to HBase.

This paper is structured as follows: in section 2 we explain our preliminaries. Section 3 provides some more detail with regards to traceability within M2M transformation. We then illustrate the shortcomings in section 4. In section 5 we present a summary of our solution and in section 6 we fully describe our new version of SiTra. In order to evaluate our work, we present a case study in section 7, specifically looking at transforming a non-trivial model of a relational database into HBase [22] (a NoSQL database). Followed by a couple of important points regarding how this can be implemented within ETL section 8. We then display the current related work in section 9 and conclude in section 10.

2 Preliminaries

2.1 Model Transformation Frameworks

Model transformation software tools, commonly known as model transformation frameworks, are used to execute M2M transformations [2,7,15,16,18,24]. These tools use a wide range of technologies and differ in the degree of support they provide and their complexity. Some model transformation frameworks have strong GUI support for programming, support of persistence and management of models, re-factoring checking, etc. However they all support the *core functionality* depicted in Figure 1 [8]. Each model transformation framework requires metamodels of both the source and destination and a set of transformation rules as input. Then the framework will *execute* the rules on an instance of the source

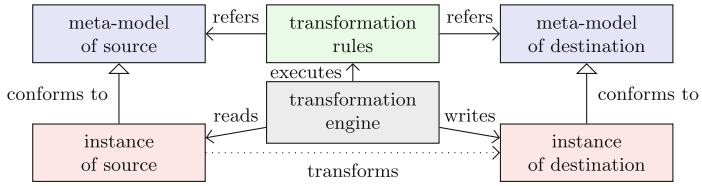

Fig. 1. An Overview of M2M Transformation

meta-model to produce an instance of the destination meta-model. In this paper we focus on this specific core functionality.

2.2 SiTra

The *Si*mple *Tra*nsformer (SiTra) is a Java library that supports the above core functionality. Produced in 2006, it has been used and modified by various groups in numerous projects and tracing activities. Among others, SiTra is used in UML2Alloy [21], AC2Alloy [13], SD2Alloy [3], OWL-S to BPEL [4], state machines to VHDL [26] and sequence diagrams to coloured Petri nets [5]. The emphasis of SiTra, although originally educational, is on using Java so that developers can execute rules in lightweight frameworks. The Java implementation is available online[1]. There are also implementations of SiTra in C# and Python.

```
1  public class ClassToTable implements Rule<Class, Table> {
2    public boolean check(Class source) {
3      return true; }
4
5    public Table build(Class source, Transformer transformer) {
6      return new Table(); }
7
8    public void setProperties(Table target, Class source,
9        Transformer transformer) {
10     List<Column> cols =
11         transformer.transformAll(source.getAttributes());
12     target.setColumns(cols);
13     ...  } ...  }
```

Fig. 2. A rule for transforming a Class object into a Table object

The framework itself defines two interfaces: *a)* `Rule`; and `Transformer`. The Rule provides an interface to create a particular output given an input and comprises of three simple methods that map to the guard, the instantiation phase

[1] http://baserg.github.io/sitra

and the binding phase of a M2M transformation. The `Transformer` interface gives the developer the bare essentials for completing an actual M2M transformation. The prime focus of SiTra is the simplicity of writing rules in an imperative language without a need of specialised tool knowledge. Figure 2 shows the popular transformation of an object orientated class to a relational table. The interface of a rule lends itself to the standard three operations within all M2M transformation engines.

1. The `check` method, line 2, is the *guard* of the rule, i.e. it determines whether the rule is applicable for the given source object.
2. The `build` method, line 5, instantiates the target object for the source that it relates to.
3. The `setProperties` method, line 8, sets the attributes of the resultant target object; from here one may call other transformations to complete the final model.

For further examples we refer the reader to the tutorial section of footnote 1.

3 Traceability within Model Transformation

Traceability is a technique for keeping track of rule invocations [18]. It has been used in many applications and has been discussed at length as an important requirement [6,11,19,23,25,27]. For a survey of traceability see "Survey of Traceability Approaches in Model-Driven Engineering" [12].

The trace instances are stored as a three tuple: $(A, AtoB, B)$. This indicates for each transformation of the source input A, using the transformation rule $AtoB$, the target output B has been created. Thus any other attempt to rerun this specific rule with the same source, the same output will be returned. This happens within popular transformation tools such as the ATL [15], QVT-O [18], the ETL [16] and SiTra.

There are however two levels of traceability: *a) internal*; and *external* as defined by [14]. Internal traceability is a private mechanism used within a transformation engine. It is used to trace what outputs are generated by what inputs. As this is internal, the API is private so the actual trace cannot be persisted and therefore is lost once the transformation is completed. ATL [15], Xtend [10] and Eclipse's implementation of QVT-O follow this mechanism. An external trace however, remains after the transformation has been completed. This enables its users to persist, or use the trace for further analysis and transformations. SiTra and ETL provide a linear trace of what rules and inputs have created what outputs.

4 Challenges of Tracing in Model Transformation

4.1 Orphans

Orphan objects are objects that are created within the M2M transformation but are not recorded within the trace. In hybrid/imperative engines like ETL and

SiTra it is possible to use the `new` keyword to create objects within the rule itself whilst not as part of the definition. Hence orphans are not accounted for within the trace, meaning if one were to attempt to find the source of this object there is no link internally or otherwise.

To see this, consider the well-known example of mapping object orientated models to relational database. This example used by Epsilon's own OO2DB example[2] the rule `Class2Table` has a conditional statement to determine whether it requires a foreign key to reference a parent table. Here it will create a `Column` and a `ForeignKey` object; neither of these are recorded within the trace due to the use of the Java allocation and not by the transformation engine.

Of course, in the above example, the ETL code can be re-factored to avoid using this keyword by using the language's ability to implement inheritance between rules. This would entail three rules: *a)* an abstract rule containing the basic `Class2Table` transformation without the if statement; a concrete, empty, rule for Classes that do not extend another; and another concrete rule for Classes that do, which extends the abstract to include the new elements for the foreign key.

In the case of SiTra, due to the restrictions placed upon in Java, the definition of a rule must only have one input and one output, i.e. `Rule<Input, Output>`. A fix for this could be the use of tuples as the `Output`, for example a `Pair<X,Y>` or `Triple<X,Y,Z>`.

The two solutions we provide here do not stop the developer from using the `new` keyword and both can increase the complexity of the rules themselves. It is not possible to remove the `new` keyword entirely. As a result, there is a clear scope in modifying the execution engines within the transformations frameworks to take good care of the orphans.

4.2 Ordering of Rule Execution

For the maintenance and debugging of a M2M transformation, the developers need to recreate the transformation. Often when a set of transformation rules is executed, there is a possibility that they are executed in a different order. This change of order can be because of the low-level implementation choices such as how a *collection* is implemented or details arising from the scheduling within the execution environment. To demonstrate the variation in the order of executing rules consider the OO2DB example used by various rule engines.

Suppose R_1 and R_2 represent the two rules that map classes and attributes to tables and table columns, respectively:

1. The Class is associated to a collection of Attributes. The overall transformation requires the `ClassToTable` to transform the attributes during its binding phase to generate the columns and assign their parent to the resultant Table object. However not all iterators iterate objects in the same order

[2] https://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.oo2db

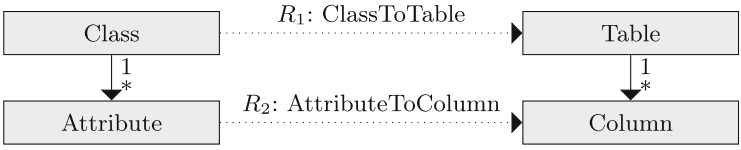

Fig. 3. A sample of rule dependencies

of which they were added. A `HashSet` in Java for example provides no guarantees as to the iteration order of the set. Thus a second execution may result in a contrasting order of elements.

2. The Starting Object: The item that *kicks off* the transformation may also change the resultant model. Given a Class *Person* with three attributes, *name*, *age* and *height* one may not assume the resultant transformation starting with `ClassToTable` would be equal to one starting with the *age* attribute using `AttributeToColumn`. If the `AttributeToColumn` were to set an order attribute within the Column it creates using a global variable which attribute is transformed first makes a difference to the final model.

To study the execution of the transformation it is essential to capture the correspondence between the source and destination elements as a part of tracing. In addition, we propose existing tracing mechanisms to be extended so that the ordering of the execution of the rules can also be captured. This would allow the developers to study the transformation, using the execution traces, and hence know in what order the rules are executed.

4.3 Rule Dependencies

Consider the example in Figure 3, which involves two dependent rules, R_1 and R_2. The execution of the transformation via ETL consists of two stages. The first step, initialisation, matches each rule to a specific source model element and creates the target elements. For example if the source meta-model consists of one class (c_1) associated to five attributes ($a_1, \ldots, a_5$), R_1 is executed once on the class to produce the table and subsequently R_2 is executed five times to produce the columns. Once the destination objects have been created, the second phase, called binding, runs the body of the rule on the objects that have been created. This part sets properties and creates the associations between the, currently disconnected, destination model elements. This is the same for ATL and QVT-O. The procedure for SiTra however is slightly different. All rules are called lazily, i.e. objects are created when they are called by parent transformations and not before. R_1 would iterate through the classes attributes to call R_2 to retrieve columns and would bind them to its table when instructed. Since ordering of the source is arbitrary, it is possible that an attribute object is processed first by R_2. This would result in starting the transformation on the attribute and that rule transforming the class that is associated to it, i.e. the

execution of R_1 on the parent class. Then from R_1 the remaining attributes are transformed, i.e. the four remaining executions of R_2.

A linear trace dictates what was created in relation to the execution of the instantiation phase. For instance a trace $T := t_1 t_2 \dots t_n$ explains that the transformation t_2 was instantiated after the transformation t_1 and that is when the targets were created. However it ignores the nested nature of a model transformation. The links between rules are lost, i.e. we don't know what rule depends on the output of another. Did, for example, t_1 require the results of t_3, or in imperative languages did t_1 transform the source of t_3 to get the results. A model transformation is a graph and this graph is lost within the standard trace. Using our example the output of $R_1(c_1)$ invoking $[R_2(a_1), R_2(a_2), \dots R_2(a_5)]$ may not be the same as $R_2(a_5)$ invoking $R_1(c_1)$, which subsequently invokes $[R_2(a_1), R_2(a_2), \dots R_2(a_4)]$. In frameworks where a developer may have a global state: it must be part of the engine to assume there is one.

Using M2M transformations to assist in a software development process, i.e. partially generating design models from architecture models, architecture models from analysis models, or generating tests from requirements, would require that the traceability be retained for auditing reasons. This is especially true for safety critical applications and in general compliance matrix generation. There is a clear scope for identifying methods of capturing the ordering of execution and inter-rule dependencies for assisting load balancing, profiling and validation.

5 Sketch of the Solution

In the previous section we have outlined some of the shortcomings of the existing tracing mechanisms in use within M2M transformations. To summarise, we extend an existing framework to deal with:

1. Capturing the nested nature of a transformation
2. Capturing rule dependencies; and
3. Capturing orphan objects created within the transformation.

Our solution involves a new meta-model to capture more information regarding the internals of a M2M transformation and the use of dynamic proxy classes to capture orphan objects. The suggested methods are independent of the model transformation frameworks and with minor alterations can be adopted by all mainstream frameworks. We explain parts of the solution briefly and then demonstrate, by case study, using a non-trivial example of transforming a relational database to HBase.

6 SiTra

6.1 Capturing Rule and Transformation Dependencies

The Simple Transformer (SiTra) is an imperative Java implementation of a M2M transformation [2]. It provides two interfaces that can be used to create a transformation engine and the rules for it. Additionally, the bundle comes with an

engine that can be used out of the box. Seyyed M. A. Shah et al. amended this to add traceability [21]. However this, like others, has all of the issues we have discussed in the previous section regarding traceability. In this section we will discuss the changes we have made to solve these issues.

We have already discussed the initialisation and binding phases within M2M transformation engines. In SiTra the initialisation phase is synonymous to the `build` method and binding is the `setProperties` method; however the scheduling differs from more declarative engines as they are called lazily rather than upfront. These two are distinct as they allow nested transformations. If you were to call a transformation which is dependent on itself, the initialised objects need to be available to the lower transformations. For instance the transformation of an *Attribute*, from a object orientated (OO) model, to a *Column*, from a relational database view, would require access to the newly transformed *Table* to set its owner. Without this we would have an infinite loop. We illustrate this inter-rule dependency with our `ClassToTable` rule, shown in Figure 2 and with a cut down `AttributeToColumn` rule shown in Figure 4. Both transformations call upon each other in order to set references.

```
1  public class AttributeToColumn implements Rule<Class, Table> {
2    public void setProperties(Column target, Attribute source,
3        Transformer transformer) {
4        Table parent = transformer.transform(source.getParent());
5        target.setParent(parent);
6      ... } ... }
```

Fig. 4. An example of an inter-rule dependency

Whilst exploring this we also found that SiTra would only transform a source object, A, once. This was because another structure is being queried within the engine, a cache. However the source object was used as the key of the map (`Map<Source, Target>`). This behaviour was found in ETL as well. Using the `equivalent()` method or `::=` operator seemed to return the first item within the transformation trace. You were able to transform a source object multiple times but if a later match was required a manual filtering of transformed objects is needed. Here we found the internal tuple needed to be amended. The map $(A, AtoB) \rightarrow B$ uses the source object, A, and the rule, $AtoB$ as the key, this allowed us to request any transformation of A given a rule with ease.

The largest issue we have found within transformation traces is the verboseness of the trace itself. A lot of information is lost when these elements are created. The current state-of-the-art provides a chronological list of rules; we never get to see the dependencies between rules and transformations. ETL uses an equivalent structure as SiTra's `ITrace` interface [21], inferred from the QVT standard [18]. In which is contained the tuple as described in section 3. This tuple does not take into consideration the nested nature of a transformation, and only concerns the instantiation phase on the first run. It may also be important to

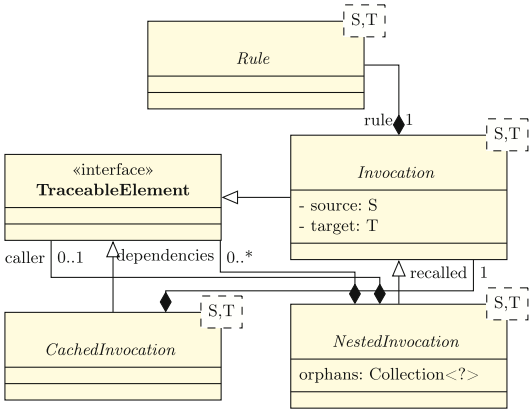

Fig. 5. A new meta-model for a traceable model transformation

see what transformations actually call the `build()` method and which have the object returned from the transformation trace. Fig. 5 shows the new transformation trace within SiTra. Here we have illustrated new types of trace element. `Invocation` is the equivalent of the previous `ITrace`, it simply contains the source, target and the rule responsible. This alone can provide the current state-of-the-art. We have introduced two more types of traceable element within SiTra: *a)* `NestedInvocation`; and `CachedInvocation`. These provide more detail to the actual internals of the transformation. The former provides the same information as the standard however contains two more elements: *a)*the calling transformation trace element (if applicable); and the trace elements generated because of the current transformation. The latter simply provides the trace element that represents the first run of the transformation. In order to maintain this list, and to reduce the effect on performance by traversing it, we amended the internal cache once more: $(A, AtoB) \rightarrow (B, TraceableElement)$. Using this latest implementation we can now see that a source element, A, and a rule, $AtoB$, returns the target object B and is referenced by $TraceableElement$. This can be simplified further as the `TraceableElement` includes B: $(A, AtoB) \rightarrow TraceableElement$.

Our meta model provides solutions to retain the order of execution of transformation rules and the ability to recreate the transformation. This is provided by the nested nature of our meta-model as it explains what rules are completed and what invoked them. The capability to find the actual binding phase, opposed to a recollection, is provided by the new cached invocation type. Allowing the user to recreate the situation at the time of creation. This cached invocation aids in providing a graph of rule dependencies.

6.2 A Dynamic Proxy to Catch Orphans

The process of creation, specifically the binding phase, involves invoking mutator methods to change the state of the destination object. These setters are often

passed new objects that need to be traced, particularly in SiTra, newly allocated objects. In order to catch these orphans we need to intercept *all* mutators to check to see if the additional objects are within the trace. For example, when adding a foreign key to a child table, we need to intercept the list of constraints.

For each transformation of source s we get a target t. In order to intercept we make a $Proxy(t)$ that maintains the functionality of the original target however the setters are modified. In each setter we check to see if the additional element is within the trace, and if not we add it the current invocation of the trace. Once this has been completed we then call the *actual* setter of the target object. To ensure traces are added for all orphans, as well as grandchildren of the target, instead of passing the original parameter we pass a proxy of it. This allows the recursion of the orphan tracking.

There are two types of call to intercept:

1. Mutator methods: we define a mutator method as one that has no return type, one parameter and begins with "set". This allows us to catch local attributes to the target.
2. Getter methods that return a collection: we define a getter as a method that has no parameters and returns a collection.

The former we have explained, however the latter is slightly different. Rather than intercepting simple *set* and *get*, we intercept collection mutators like *put*, *add*, *addAll*, etc.

7 Case Study

In order to demonstrate our new framework we have created a non-trivial transformation between a relational database and HBase, involving a transformation of the schema as well as the data. We then applied this to an *instance* of the relational database using the employee database provided by MySQL[3], a widely used test database for benchmarking. This dataset contains four million rows over six related tables.

For the purpose of this case study, and due to space restraints, we shall not delve into the rules in depth for this transformation. Instead we shall use them, partial or otherwise, to demonstrate what happens within the black-box that is SiTra. The M2M transformation itself will be available online[4].

Meta-Model of Apache HBase. The meta-model of Apache HBase, the destination, is shown in Figure 6. Here we can see a very simple representation of the internals of the NoSQL database engine. We have a Namespace, which is synonymous to a Database in relational terms, but that is as far as similarities between HBase and a relational database go. A Table in a NoSQL sense is more of a key-value store, whereas the relational view would view its Tables as a tree

[3] https://dev.mysql.com/doc/employee/en/
[4] https://baserg.github.io/sitra

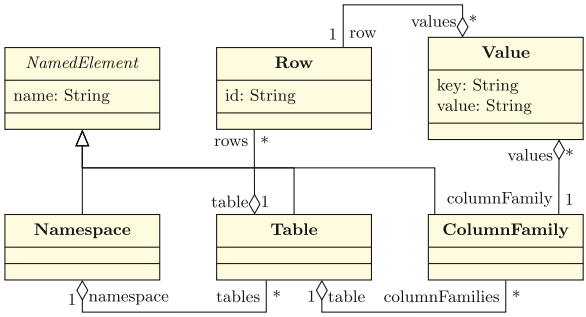

Fig. 6. The meta-model of a Apache HBase

structure. A Table contains a selection of Column Families and Rows. The former enables more structure within the key store whilst the latter is the data itself. A Row contains an id, this is the key for all related data to this Table. Finally we have Values tied to the Rows and Column Families, each value has its own key to differentiate itself from the other values within a Column Family.

This meta-model allows us to realise the structure and the data of Apache HBase. In turn this is used to generate HBase shell to persist our transformation to a real HBase server. In order to do this we use the template engine Xtend.

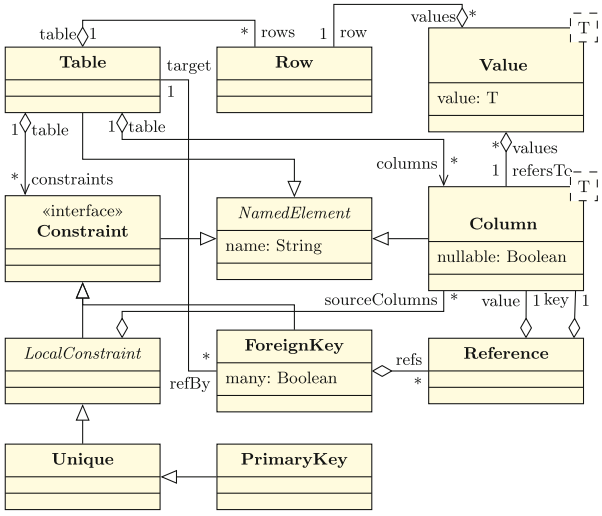

Fig. 7. The meta-model of a relational database

Meta-Model of a Relational Database. Since NoSQL databases generally do not have schemas, we needed a meta-model for a relational database that includes the

data itself. Unlike OO2DB where we are purely transforming structure, NoSQL database avoids creating structure unnecessarily. The primitive structures that are created are simple buckets for string or binary data. Therefore in order to properly transform a database we need to access the data within the relational tables.

Figure 7 shows us the meta-model of a relational database, our source meta-model. Here we have a Table with Constraints, Rows and Columns. Where a Constraint is local to the Table, i.e. a Unique index, a Primary Key or a Foreign Key. Those in turn reference Columns to enforce their Constraint and in the case of a Foreign Key provide a mapping of a selection of Columns of one table to another target table. In order to keep the data, we have a Value class, which references the Column it belongs to and the Row it is part of. This latter element allows us to have the data we require for the transformation.

7.1 Catching Orphans

In order to depict the orphan issue we take a subset of the transformation we have created. Specifically the transformation of a *Relational Table* to a *HBase Table*. When creating a HBase Table one must make a *default* column family to hold the primitive data. For example the `employee` would contain a column family called *0* and in there would have values relating to their name, age and gender. In SiTra we would define a rule to be `Rule<db.Table, hbase.Table>`, this would only execute on tables with no or more than two foreign keys, as this would be a *root* table or a complex lookup table (which would need to be referenced).

We can envisage a binding phase as shown in Figure 8. As normal, the `target` would appear in the internal trace of SiTra as it would be added after the instantiation phase, however we have introduced a new element: a column family. This element is disconnected from the transformation trace. However when `setProperties` is called, the `hbase.Table` is in fact a dynamic proxy instance. This instance, as mentioned in subsection 6.2, captures *getters* whereby the return is a collection and in turn returns a collection proxy, which intercepts the lists mutators. Before the addition to the collection is made, the proxy determines whether it has seen the `columnFamily` before and if not adds it to the *orphan* collection in the currently active trace instance (as seen in Figure 5).

```
public void setProperties(hbase.Table target, db.Table source,
    Transformer transformer) {
  ColumnFamily columnFamily = new hbase.ColumnFamily();
  columnFamily.setName("0");
  target.getColumnFamilies().add(columnFamily);
}
```

Fig. 8. A sample of a scenario leading to The creation of an orphan in SiTra

Our transformation of the employee database manually creates columns families in the same fashion as above, both the employee and department tables have this default column family. Relationships however are treated slightly differently. Those lookup tables aren't transformed into different tables, instead are a group of column families attached to the parent table. Therefore each relational child table, the columns are converted to column families and are added to the parent HBase table. This still uses the same mechanism as above, however there is a loop to iterate the new column families. SiTra was able to retain all orphan objects for this transformation.

7.2 The Nested Nature of Model-to-Model Transformation

Continuing the example of a relational table and an HBase table, the transformation will recurse by transforming the rows that the table has, and in turn the values will be transformed. This natural tree structure that happens is captured within the new meta-model. Whereby an invocation *depends* on another, as shown in Figure 5. In order to implement, and retain this information, we use a simple stack. Not unlike a process stack, our *stack frame* is the invocation element as it has access to all components: source, target, orphans and of course dependencies. When an item is *built* a trace element is created to record this transaction, it is then added to the top of the stack. Once the binding phase has completed, it is then popped off the top.

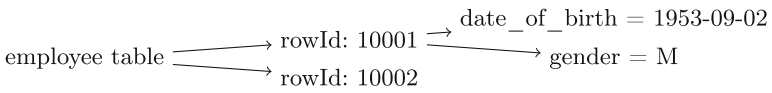

Fig. 9. Depicting the nesting of rules

Figure 9 illustrates a small portion of our output model. Level one is that of a relational table to an HBase table, level two transforms the rows, where the `rowId` is the primary key of the table. Finally level three is the transformation of the values, for the default column family. The tree for the whole transformation is very large; however we have an implementation that can persist these links into the graph database Neo4j (available at http://www.neo4j.org).

7.3 Deriving Rule Dependencies

Our meta-model retains the information regarding all transactions within the transformation, particularly the recollection of previously transformed elements of the source model, which is currently unavailable from the current state-of-the-art. These two relationships between different transformations can be used to derive the inter-rule dependencies. For example, when transforming a relational table to a HBase table, the rule will attempt to transform its data, i.e. its rows.

Once this is complete it will add the rows to itself. However the rows themselves require the HBase table to add itself to it, its opposite. This cyclic assignment is a must if we do not have a modelling framework to automatically set these links, like ECORE. From this point we can derive that the first rule, in this instance, depends on the second, and vice versus.

To gather this we need only iterate through the trace elements and generate a graph of the rules used. If we move down a level, of `NestedInvocation`, we know that the parent required it, if we find a `CachedInvocation` we know *a)* it has been transformed before; and it has been recalled for this current execution.

8 Epsilon Transformation Language

The mechanisms described in this paper can be applied to other frameworks as well. The meta-model described in Figure 5 can be presented in most frameworks; however a key difference with SiTra and ETL is the scheduling. ETL flattens the model, matches and instantiates all model elements before binding them whereas SiTra is completed on the fly. However ETL can derive the dependency links between transformations by realising the first time the bind is called on an object, and by the way transformations are referenced, using the `equivalent()` methods. The orphan capture can also be completed: if using ECORE one may use its native notification pattern. `EContentAdapter` can be used on either the first transformation or upon the ECORE resource that is tracking the output. Opposed to intercepting calls, as is needed on regular POJOs, one may simply interpret the notification from the change methods.

9 Related Work

Mäder explains that traceability links are rarely re-used in the maintenance of a system despite the ever-increasing complexities that they contain [17]. He puts partial blame to the failure of tools to provide usable functionality for stakeholders to query and capture traceability links. The move to an integrated traceability mechanism with a verbose trace would allow it to be persisted inside a data store such that standard queries can be made in an attempt to solve some of those issues.

Frédérick Jouault argues that traceability need not be part of the overall transformation engine as a High Order Transformation (HOT) can be used, he uses HOT, with an ATL example, to introduce trace link elements into an existing transformation script [14]. Here an instance of ATL is transformed into another version of ATL with additional outputs along with an imperative binding. Iván Santiago et al. also used this in order to add iTrace capabilities to transformations so they can measure the quality degradation that the introduction of trace generation causes [20]. Here we have a decoupled the mechanism used to implement traceability; however this is an additional step for validation. Our approach maintains the implementation within the framework in order to remove the additional burden it places upon the developer.

10 Conclusion

The primary conclusion of this paper is that there is little in terms of built in traceability in rule-based transformation engines. Those that do provide an external trace are unable to provide enough information to relate to the internals of a M2M transformation. Model transformations themselves are relational processes, they relate the parties involved: sources, targets and rules, but also relate to each other and generate a dependency model within rules and executions of rules. The latter is lost in QVT's trace instance.

We have provided a new, independent, trace meta-model that could be used within most M2M transformations engines to maintain the tracing information. In addition we have provided some information on how other engines may implement this functionality, particularly the ability to track orphans. To demonstrate these mechanisms we have implemented it by extending SiTra. These changes to SiTra have brought it up to the current state-of-the art in terms of traceability and provide these mechanisms natively.

References

1. Aizenbud-Reshef, N., et al.: Model traceability. IBM Systems Journal **45** (2006)
2. Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra: simple transformations in java. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 351–364. Springer, Heidelberg (2006)
3. Alwanain, M., Bordbar, B., Küster, J., Bowles, F.: Automated Composition of Sequence Diagrams via Alloy. MODELSWARD (2014)
4. Bordbar, B., Howells, G., Evans, M., Staikopoulos, A.: Model transformation from OWL-S to BPEL Via SiTra. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 43–58. Springer, Heidelberg (2007)
5. Bowles, J., Meedeniya, D.: Formal Transformation from Sequence Diagrams to Coloured Petri Nets. In: 2010 17th Asia Pacific Software Engineering Conference (APSEC) (2010)
6. Briand, L., et al.: Traceability and SysML design slices to support safety inspections: a controlled experiment. ACM Trans. Softw. Eng. Methodol. **23** (2014)
7. Claypool, K.T., Rundensteiner, E.A.: Gangam: a transformation modeling framework. In: 2003. (DASFAA 2003) Proceedings Eighth International Conference on Database Systems for Advanced Applications (2003)
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45** (2006)
9. Ebner, G., Kaindl, H.: Tracing all around in reengineering. IEEE Software **19** (2002)
10. Eclipse Foundation. Xtend (2014). URL: http://www.eclipse.org/xtend/ (visited on 03/04/2015)
11. Fritzsche, M. et al.: Application of Tracing Techniques in Model-Driven Performance Engineering. In: 4th ECMDA Traceability Workshop (2008)
12. Galvao, I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: 2007 EDOC 2007 11th IEEE International Enterprise Distributed Object Computing Conference (2007)

13. Geepalla, E., Bordbar, B., Last, J.: Transformation of spatio-temporal role based access control specification to alloy. In: Abelló, A., Bellatreche, L., Benatallah, B. (eds.) MEDI 2012. LNCS, vol. 7602, pp. 67–78. Springer, Heidelberg (2012)
14. Jouault, F.: Loosely coupled traceability for ATL. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) Workshop on Traceability (2005)
15. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
16. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
17. Mäder, P.: Interactive traceability querying and visualization for coping with development complexity. In: CoRR (2013)
18. OMG. Meta Object Facility (MOF) 2.0 Query View Transformation Specification Version 1.1., Jan. 2011. URL: http://www.omg.org/spec/QVT/1.1/PDF/ (visited on 03/04/2015)
19. Paige, R.F., et al.: Building model-driven engineering traceability classifications. In: 4th ECMDA Traceability Workshop (2008)
20. Santiago, I., Vara, J.M., de Castro, V., Marcos, E.: Measuring the effect of enabling traces generation in ATL model transformations. In: Filipe, J., Maciaszek, L.A. (eds.) ENASE 2013. CCIS, vol. 417, pp. 229–240. Springer, Heidelberg (2013)
21. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to alloy and back again. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 158–171. Springer, Heidelberg (2010)
22. The Apache Software Foundation. Apache HBase (2014). URL: http://hbase.apache.org/ (visited on 03/05/2015)
23. Vara, J.M., et al.: Dealing with traceability in the MDDof model transformations. In: IEEE Transactions on Software Engineering **40** (2014)
24. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming **68** (2007)
25. Willink, E.D., Matragkas, N.: QVT Traceability: What does it really mean? (2014). URL: http://www.eclipse.org/mmt/qvt/docs/ICMT2014/QVTtraceability.pdf (visited on 03/04/2015)
26. Wood, S.K., et al.: A model-driven development approach to mapping UML state diagrams to synthesizable VHDL. IEEE Transactions on Computers **57** (2008)
27. Yie, A., Wagelaar, D.: Advanced Traceability for ATL. In: Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL 2009) (2009)