

QVT Traceability: What does it really mean?

Edward D. Willink¹ and Nicholas Matragkas²

¹ Willink Transformations Ltd., UK,
`ed-at-willink.me.uk`

² University of York, UK,
`nicholas.matragkas-at-york.ac.uk`

Abstract. Traceability in Model Transformation languages supports not only post-execution analysis, but also incremental update and co-ordination of repetition. The Query/View/Transformation family of languages specify a form of traceability that unifies high and low level abstraction in declarative and imperative transformation languages. Unfortunately this aspect of the QVT specification is little more than an aspiration. We identify axioms that resolve the conflicting requirements on traceability, and provide a foundation for resolving further issues regarding equality, transformation extension and mapping refinement.

Keywords: Model Transformation, QVT, Traceability, Transformation Extension, Mapping Refinement

1 Introduction

The successes of early model transformation prompted the Object Management Group (OMG) to issue a Request for Proposal for a standard solution to the problems of querying, viewing and transforming models. The initial submissions eventually converged to give us the Query/View/Transformation (QVT) family of languages. The specification standardizes research work and so the specification is imperfect.

Problems in the specification are recorded as issues that can be raised by anyone at http://www.omg.org/report_issue.htm. The issues are addressed by a Revision Task Force (RTF) that proposes and votes on resolutions. The QVT 1.2 RTF has just completed resolutions for the 125 outstanding issues against QVT 1.1[9]. At the time of writing 2/3 have been balloted successfully, with the remaining 1/3 being voted on. However 20% of the resolutions just agree to defer difficult issues for further consideration. This paper provides some of the further consideration and an opportunity for the wider community to comment on these issues, which while described in the context of QVT, are relevant to model transformation more generally.

In Section 2 we first summarize the specified aspects of QVT traceability, then in Section 3 we dig down to the underlying purpose of traceability in a declarative context and formulate some axioms that enable us to consider the Object-Orientedness of transformations. The more challenging context of Imperative Transformations is discussed in Section 4. In Section 5 we outline a Trace

Model that can be used more realistically throughout QVT. We present related work in Section 6 and finally conclude in Section 7.

2 QVT Traceability as Specified

The QVT specification defines a family of three languages, in part reflecting differing viewpoints amongst the original submitters and in part reflecting different community requirements.

- QVT Operational Mappings (QVTo) an imperative transformation language.
- QVT Relations (QVTr) a high level declarative transformation language.
- QVT Core (QVTc) a low level declarative transformation language.

These languages are unified via the QVTc semantics. QVTr transforms directly to QVTc. QVTo and arbitrary externally supplied blackboxes are defined with respect to an equivalent QVTr Relation. The practical compatibility of this unification has yet to be demonstrated, since the various QVTo and QVTr vendors find sufficient challenges in realizing a primary language and blackboxes without worrying about the alternative paradigm. There are no complete QVTc implementations.

In this paper we concentrate on a narrower and potentially much clearer form of compatibility; the *trace records* which are supposedly common to all three languages.

One benefit and consequence of the low level QVTc language is that the user can and must specify the maintenance of the classes that establish traceability. These form the middle model for a typical bidirectional QVTc mapping in which left and right hand models are related with the aid of the intermediate trace model.

A `Package2Schema` mapping may therefore operate between a `Package p` and a `Schema s` maintaining a *trace record* of the intermediate `TPackage2Schema trace class` with `p` and `s` properties.

```
class TPackage2Schema {
    property p : Package;
    property s : Schema;
}
```

When the mapping is executed in the forward direction, the `p` property is initialized with the matched input `Package` and the `s` property with the generated output `Schema`. The *trace instance* records the `Package2Schema` relationship between `p` and `s` when the source-to-target relationship is first established. Thereafter another mapping can locate the schema corresponding to `aPackage` by the OCL navigation `aPackage.TPackage2Schema.s`, which starts at `aPackage`, uses the `Package::TPackage2Schema` relationship, which is the implicit opposite of the `TPackage2Schema::p` relationship. This locates the *trace record* and then

the `TPackage2Schema::s` relationship navigates to the target. This is a demonstration of the power of an explicitly modeled *trace class*; there is no need for any special operators such as `resolve` in QVTo or `resolveTemp` in ATL.

The need for manual maintenance of the *trace records* is eliminated in QVTr for which a mapping to QVTc is suggested by Rule 1 of Clause 10.2 of the QVT specification. Unfortunately this rule neglects to specify an algorithm for construction of the trace class name and so there is no guidance on how name clashes are to be avoided.

Defining *trace classes* without any common base class to support polymorphic *trace instance* maintenance makes this aspect of the QVT specification totally unsuitable for practical tooling. In Section 5 we outline a more rational proposal.

The specification of QVTo *trace classes* is distinctly vague since it relies on the underspecified correspondence between a QVTo MappingOperation with an unspecified Operation signature, and a QVTr Relation. Overloading is even more of a problem in QVTo with its disjuncted mappings. The need to extend the *trace data* to accommodate *in/inout/out* parameters is identified without any clarification on how the trace data is modified in a way that remains compatible with QVTc. *inout* parameters present a further challenge; in QVTo an *inout* parameter has distinct *in* and *out* values. This is incompatible with a one-to-one correspondence with a QVTr relation. QVTr is multi-directional, so a parameter may be *in* or *out* depending on the direction; it cannot be both *in* and *out*.

3 Declarative QVT Traceability Purpose

Traceability is useful to analyze the behavior of the transform when debugging or tuning, but much more importantly it enables a model transformation to transform to a graph rather than a tree. When building cross-references in the output graph, it is necessary to reference a target model element and this requires an ability to navigate the source-to-target relationships proved by the *trace records*.

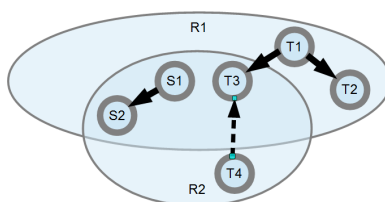


Fig. 1. Target resolution Problem

Fig 1 shows the basic problem. We know that mapping R1 establishes a relationship between pairs of source model elements S1, S2 and triples of target model elements T1, T2, T3. In another mapping R2, where a similarly typed pair of source model elements S1, S2 reoccurs, we require to re-reference one of

R1's target model elements, T3, in order to establish a cross-reference from T4. We can solve this problem if we maintain *trace data* comprising a *trace record* that relates the tuple (R1,S1,S2) to the tuple (T1,T2,T3). We then just ask the *trace data* to return the appropriate (T1,T2,T3) for our new relationship R2 where we want to know about (R1,S1,S2) model elements.

We will now examine what properties we need traceability to exhibit in order to perform declarative transformations. Once we understand declarative transformations, we can then see in Section 4, how the QVTo imperative transformation language can conform to the same principles.

3.1 Trace Record Axioms

Trace Representation A *trace representation* summarizes the state of an input or an output model element.

The *trace representations* of any pair of model elements contains sufficient information to determine whether the two model elements are equal. Additionally an *output trace representation* contains sufficient content to allow the represented element to be reconstructed.

UML, and consequently OCL and transformation languages based on UML or OCL, define two importantly different forms of model element.

Each Class instance or object represents a distinct model element; no two objects are ever equal. It is therefore convenient to summarize the state of any object by its memory address since each object has a distinct address. Reconstruction of the object from the summary is trivial, so long as the object remains in memory.

An instance of a DataType (such as Integer or a Collection(String)) has a value. Instances of DataTypes are equal when they have the same value, even if the instances are stored at distinct addresses. 4 is clearly always equal to 4 no matter what address the 4 is at. For values such as Integers, whose storage requirements are small, it is convenient to use the entire value as the representation. For potentially very large values such as Strings or Collections, it is convenient to use the address as the summary, making reconstruction easy, but requiring care to compare the underlying value rather than just the *trace representation* when determining whether two DataType values are equal.

Trace Record A *trace record*, which may also be called a *trace instance* of a *trace class*, comprises

- the identity of the mapping that is traced
- an ordered list of the *input trace representations* of each mapping input argument
- an ordered list of the *output trace representations* of each mapping output result

(The ordering of the lists may be established by some deterministic algorithm such as an alphabetic sort applied to the names of the mapping parameters.)

Trace Identity A *trace identity* comprises the first two elements of the trace record; mapping identity and *input trace representations*.

Trace Identity Uniqueness No two *trace records* share the same *trace identity*.

Trace Data The *trace data* comprises all the *trace records*.

Trace Record Creation Every mapping execution has a *trace identity* and a corresponding *trace record*.

Consequently any attempt to execute a mapping that would require a duplicate *trace identity* must be detected and suppressed. The results of the suppressed execution are reconstructed from the *output trace representations* of the previous execution.

Trace Record Access The *trace data* may be exploited to recover a *trace record* for a given *trace identity* and so locate the target for a graph-to-graph transformation.

Consequently the *output trace representation* must be consistently determined by the *trace identity*.

If an underlying input model element changes, then the *input trace representation* changes too and so the mapping must be re-executed if the corresponding outputs are required.

3.2 Tracing immutable Class instances and DataType values

With distinct input and output models, the input model elements may be stable for a declarative transformation or a well-behaved imperative transformation and so the use of an address to summarize each unchanging object is valid.

Maintaining a *trace record* for immutable `DataType` or `Class` instances therefore just requires the address of the relevant instance to be retained. This incurs very limited cost apart from limiting Garbage Collection in long running programs.

3.3 Tracing mutable Class instances and DataType values

However for an in-place transformation or less well-behaved imperative transformation, model elements may be updated requiring a more complete *input trace representation* to ensure that the mapping is executed again for a changed input to yield a correspondingly changed output.

For in-place declarative transformations, it may be possible to treat the transformation as an out-of-place transformation followed by a copy back to the input so that all changes are localized in the copy-back.

3.4 Child Stealing

Access to previous results is necessary to be able to resolve the targets in graph-to-graph transformations. The declarative formulation, as replacement of the result of a mapping re-execution by the original results, can have an unpleasant corollary if those graphs are UML or Ecore models with composition relationships. A composition relationship imposes a limitation that a model element can only be in one container. Consequently a conflict arises when a model element, that already is already the target of one composition relationship, is assigned as the target of another composition relationship.

The problem is demonstrated by the following example in which the intent is to transform an incoming model comprising a single `Node` instance, into a model comprising a `Node` instance with 10 composed child `Node` instances and a list of 10 references, one to each child `Node`.

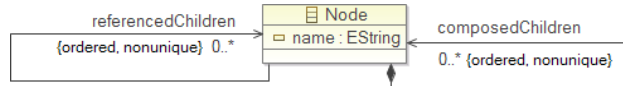


Fig. 2. Metamodel for the child-stealing example

However when we execute the transformation, we get a `Node` instance with a single child `Node` and a list with 10 references to the one child.

```

modeltype MODEL uses 'p';

transformation ChildStealing(in nodes : MODEL, out MODEL);

main() {
  nodes.rootObjects()[MODEL::Node]->map node2nodes();
}

mapping MODEL::Node::node2nodes() : MODEL::Node {
  result.name := self.name + 'AtRoot';
  result.referencedChildren += Sequence{1..10}->xcollect(i | self.map node2node(0));
  result.composedChildren += Sequence{1..10}->xcollect(i | self.map node2node(0));
}

mapping MODEL::Node::node2node(i : Integer) : MODEL::Node
{
  result.name := self.name + i.toString();
}

```

Fig. 3. QVTo Code for the child-stealing example

The critical part of the code is `Sequence{1..10}->xcollect(i` that arranges for `i` to loop over the integers 1 to 10 and collect the results of a

`self.map node2node(0)` mapping invocation at each loop iteration. The first invocation of the mapping creates a new `Node` as required, but subsequent invocations have the same `(node2node, self, 0)` *trace identity* and so return the existing `Node`. The collected results therefore comprise the same result ten times and when `result.composedChildren +=` assigns them to a composition, the first assignment succeeds and each subsequent assignment re-assigns.

Users of the Java API for EMF[10] will be familiar with the confusing disappearance of elements from containers because they were reassigned to alternative containers in seemingly totally unrelated code. For Java programmers, this may be an unavoidable consequence of low level practices. For an imperative transformation language the surprising non-local side effects are very undesirable, but can at least be predictably located when single stepping. For a declarative transformation language, there is no clear schedule and so the surprise may be dependent on the accidental ordering of mappings executions; this cannot be acceptable.

We have a conflict between traceability-driven re-use and composition; the results that the *trace record* gives us for re-use are not fit for exclusive re-use. The problem is that the *trace identity* has insufficient discrimination to exclude alternative re-use. In the example above the tuple `(node2node, self:Node, i:Integer)` is adequate but the calling code always uses 0 as the Integer provoking the unwanted re-use. Changing the caller to pass the loop ‘counter’ to the mapping makes the usage exclusive and we get the desired 10 `composedChildren`.

We can see that Child Stealing is a serious problem and we can easily detect it. During normal execution each model element is created without a container and then assigned to a container. It is very rare for model elements to be intentionally re-assigned to another container, so we can define any such re-assignment as a Child Stealing Error. If there is a real requirement for re-assignment, we can require that it is performed in two stages, first to explicitly de-assign the container before re-assigning. In some cases it may be possible to detect Child Stealing Errors by static analysis.

3.5 Transformation Extension and Mapping Refinement

All three QVT languages define transformations that may extend other transformations, and mappings that may refine other mappings, but neglect to define what this may mean. The immediate question is are transformations Object-Oriented and do they observe the Liskov Substitution Principle[7]?

We would very much like transformations to be Object-Oriented so that we can use familiar derive and override practices, so let us assume a strong analogy between the new Transformation/Extension/Mapping/Refinement concepts and the familiar Class/Inheritance/Operation/Overriding. We will exploit the traceability axioms to see what limitations we may need.

The Liskov Substitution Principle is of course never observed unequivocally, but disciplined transformation writers can define those behaviors for which substitutability is important and code their refinements accordingly.

Transformation Extension For the model parameters of an extended transformation, we might look for an analogy with the arguments of a derived constructor, where the arguments can be very different and must be converted to use one of the inherited constructors. However for transformations there is just a single interface of one model-type per model-parameter and there are no conversion facilities. A model-type is a list of packages of types that the transformation compiler needs to consider; the model-type is not significant at run-time. The transformation extension therefore just routes the models of the extending transformation to the extended transformation. Since the execution direction is unknown, every type that could be generated by the extended transformation must be included by the model-type of the extending transformation, so we require:

All extending transformation model-types must be super-sets of the extended model-types.

Mapping Refinement The declarative QVT languages are multi-directional and so we do not know which mapping parameters are inputs and outputs. There is therefore no opportunity for covariant or contra-variant refined mapping parameter types.

Refining mappings must overload the refined mapping signature without change.

Dynamic Dispatch For OO-style dynamic dispatch of mappings to work, we require:

QVTc and QVTr must have a this object that is an instance of the executing transformation.

The instance can be stateless.

Tracing When we consider a *trace record* involving execution of a refined mapping, should the apparent invoked mapping or the actual executed mapping be recorded? A subsequent user of the trace must be able to resolve against the apparent mapping, rather than all possible refinements, but must also be able to resolve against the actual mapping where that is known.

There must be a trace record for each possible trace identity.

This does not necessarily imply multiple *trace records* and in Section 5 we propose to allow a *trace record* to have multiple *trace identities*.

Composition An overridden declarative mapping provides a composition of predicates and assignments.

In accordance with Design by Contract[8] all predicates should be satisfied. There may obviously be scope for optimization of redundancy.

Similarly all assignments should be made, but since multiple assignments may conflict, each most derived assignment should be chosen, thereby allowing a refining mapping to adjust as well as extend the behaviour of a refined mapping.

A mapping refinement hierarchy is equivalent to a single composite mapping.

Concurrency If we have a system in which more than one copy of a transformation executes on the same input models, we may want to share common model elements but must keep composed model elements exclusive. Providing a single overall *trace data* will satisfactorily share the common model elements, but we must avoid Child Stealing of the exclusive model elements. Most mappings create composed children with the intended parent as one of the traced inputs, so provided the parents are exclusive, the children will be too. We therefore only need to ensure that the root parent is exclusive, which may be achieved by using the transformation *this* as part of the *trace identity*.

Trace data may be shared by concurrent transformations.

This same reasoning applies to the nested execution of extended/extending transformations or refined/refining mappings.

4 Imperative Traceability

So far we have concentrated on the declarative QVT languages where the ability for objects to mutate and the ability to impose a schedule is very limited. Declarative scheduling is an implicit consequence of the inability to perform a mapping until its inputs are available. Complex input/output dependencies may result in the mappings being grouped into passes. Object mutation is limited to one value at the input/output of each ‘pass’.

4.1 Imperative Characteristics

In contrast QVT allows the user to

- specify an explicit execution order
- make arbitrary changes to objects
- use mutable collections
- exploit changeable global context
- exploit changeable transformation context
- have *inout* mapping arguments

Of these only the explicit order is not a problem for traceability; it just allows an infeasible order to be programmed.

Object changes If an object changes, the result obtained from a mapping execution using that object prior to the change may not be valid and so the assumption that its state can be summarized by just its address is unsound; the *input trace representation* must be extended to all mutable fields that influence the mapping execution. This may be difficult to determine when complex helper functions are used, so it may be necessary for the *input trace representation* to create a deep-clone of the traced input unless an existing deep-clone can be re-used.

The *trace record* must also maintain a deep-clone of the output in order to avoid any corruption by user assignments, but this makes it impossible to return the same result as a previous mapping execution, since each new result must be a clone to avoid corruption by assignments.

Collection changes QVTo introduces two new forms of mutable collection: List and Dict. These are very convenient for imperative programming, but have never been fully characterized in terms of their UML alignment. Is List{1} equal to List{1}? Obviously Yes. Is List{2} equal to List{1}? Obviously Not, not even if List{2} arose from changing List{1}. List is therefore clearly a Mutable DataType whose memory location is irrelevant since only its value is interesting.

This therefore presents the same problem as for Object changes. The *input trace representation* and *output trace representation* must be deep-clones to avoid value corruption in the *trace record*.

Global context changes The global context such as configuration properties may influence the mapping execution and the global context may be changed between mapping executions. It is therefore necessary to include the prevailing state of all relevant mutable global context as part of the *trace identity* in order to accurately determine whether the previous mapping execution result can be re-used.

Transformation context changes Transformation context such as contextual properties and intermediate classes may also influence the mapping execution and again this context may be changed between or during executions. It is therefore also necessary to include the prevailing state of all relevant mutable transformation context as part of the *trace identity*.

Mapping argument changes A QVTo mapping may have *inout* arguments which may influence the behavior, so we should deep-clone the relevant parts of the value on input as one of the *input trace representations* and the whole output as one of the *output trace representations*.

4.2 Assessment

It may be noted that all this additional context is also required to fulfill the goal that the trace record contains the information necessary to determine what needs to be re-executed as part of an incremental update.

QVTo is intended to be a practical language, so introducing all the additional *trace record* content identified above to make traceability sound seems unacceptable. The cost of all the deep-cloning is obviously large and may be quadratically so. Consider a relatively simple mapping in which the programmer either does not understand, or does not trust, the built-in *trace record* resolution capabilities. An *inout* Dict is passed in order to keep track of all the input to output mappings. This Dict will grow in size with each mapping and so we have a steadily growing Dict to deep-clone and consequently $O(N*N)$ complexity.

We must instead try to tighten to the language to retain utility while adding some integrity.

The major problems of deep-cloning to stabilize the *trace identity* can be avoided if all traced inputs are transitively immutable. This can be achieved for

inputs by requiring an *in* rather than *inout* parameter and introducing a new restriction that only constants, exclusive clones or *in* parameters may be passed to *in* parameters. This guarantees that *in* can be used without deep-cloning.

A similar policy could be applied to outputs, but the language would be useless; it would be impossible to execute a mapping to create a parent object and then populate its children, either in the caller or in a separate piece of code that uses the trace to resolve the output for update in a second pass. It seems necessary to allow the *output trace representation* to evolve.

Changes to the global context can be ignored, in the hope that users will restrict the use of global context to stable configuration that requires a total re-evaluation for any change. It may be possible for tooling to detect global context sensitivity and guide users into refactoring their transformation so that unstable global context becomes a disciplined input model.

Some changes to the transformation context could be similarly ignored, but the transformation itself cannot. There is no way that two concurrent imperative transformations operating on the same models can safely share mapping execution outputs. The transformation *this* must therefore form part of an overall *trace identity* or distinct *trace data* must be maintained for each transformation execution.

inout arguments must also be ignored in the hope that their usage is to accumulate additional information that does not influence the mapping execution result. This is true of the user-maintained traceability example above. It may often be possible to verify this statically, but not always. We may perhaps be able to impose a transitive prohibition on an *in* or *out* argument ever being used as an *inout* argument.

4.3 Transformation Extension and Mapping Refinement

The declarations of QVTo's `OperationalTransformation` and `MappingOperation` have many similarities to the declarative `Transformation` and `Mapping` so we may look to achieve Object Oriented behavior for imperative transformations as well.

QVTo provides additional mapping refinement options.

A *disjunct* mapping provides an outer mapping which redirects to one of a number of inner mappings according to a type and guard-based selection. Since the outer mapping performs no object creation, the *trace record* for a disjunct mapping can be provided by the selected inner mapping, augmented by the additional *trace identity* of the also-executed outer mapping.

An *inherited* mapping creates an object in the outer mapping before the inherited inner mapping initializes it and returns control for further execution by the outer mapping. Since the outer mapping performs the object creation, the *trace record* for an inherited mapping can be provided by the outer mapping, augmented by the additional *trace identity* of the also-executed inner mapping.

A *merged* mapping executes an outer mapping then executes an inner mapping. Since the outer mapping performs the object creation, the *trace record* for

an inherited mapping can be provided by the outer mapping, augmented by the additional *trace identity* of the also-executed inner mapping.

4.4 Incremental Update

The QVT specification mentions the utility of the *trace data* to support efficient incremental update. This may be supported by QVTc and consequently QVTr, but is certainly not for QVTo as specified.

QVTo specifies that a *trace record* is created during the object initialization phase, which occurs after guards have been evaluated. Consequently there is no trace of model elements that were not created and so it is difficult for an incremental update to correctly handle a change that affects the execution of the guards.

QVTo provides no *trace record* for explicit object creation or for explicit cloning.

These limitations in conjunction with the pragmatic exclusion of global, transformation and *inout* context from the *trace record* suggest that in order to achieve reliable incremental update with QVTo, it may be necessary to impose so many declarative characteristics that it may be more profitable to develop a QVTo to QVTr migration assistant than an incremental QVTo solution.

5 Multi-language solution

In Section 2 we identified the strong bias of the QVT specification trace model to QVTc. In Figure 4 we suggest a more flexible metamodel that avoids imposing enumerated QVTc-style trace classes on QVTo, supports polymorphic access to the trace and provides for extension to support specialized tracing of specialized mappings such as QVTo disjuncts.

The abstract `TraceRecord` provides the freedom for `QVTcTraceRecord` and `QVToTraceRecord` to pursue different representation strategies.

The `TPackage2Schema` is an example of an enumerated QVTc trace class. Apart from the inheritance, it is exactly what, and all that, the QVT specification defines.

The `TraceData::getMapping` API supports discovery of a trace record for a given `Mapping` operation and list of `Elements`. The `TraceRecord::getMappings` provides the list of all `Mapping` operations executed. `TraceRecord::getInput` or `getOutput` enable a particular input or output model element to be recovered. `TraceRecord::wasExecuted` enables additional tracing of unexecuted mappings to be recorded to support incremental update.

6 Related Work

In the context of model transformation, traceability can be used in a variety of scenarios such as impact analysis or object resolution. Due to this the majority of model transformation languages provide direct or indirect support for tracing.

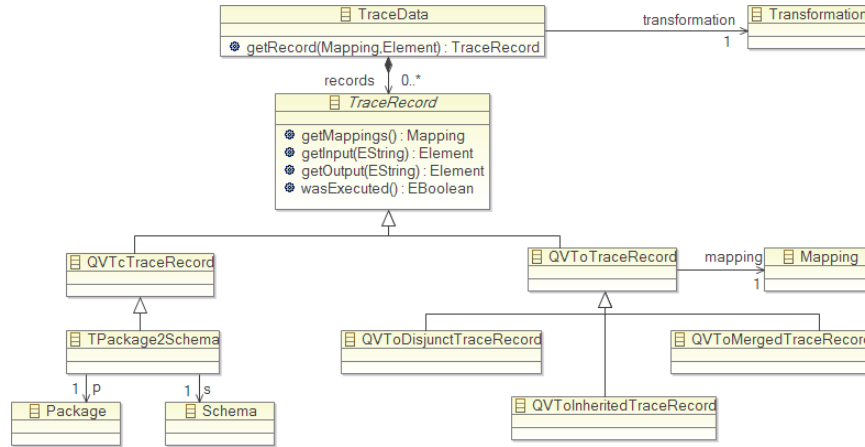


Fig. 4. Metamodel for the Trace Data solution

The Atlas Transformation Language (ATL) [5] uses tracing to resolve the interactions and dependencies between the rules of a transformation. The traceability mechanism offered by ATL is implicit, in the sense that the tracing information is captured automatically by the transformation engine without any input or guidance from the end user. Every time a transformation rule is matched, a new trace link object is created between the source element and its corresponding target element(s) by using the native type *ASMTransientLink*. This trace link is assigned the name of the rule, the source and target elements, and it is added to a link collection named *ASMTransientLinkSet*. This link collection is used internally by the ATL virtual machine and it is not directly accessible by the end user. ATL provides the *resolveTemp()* method, which makes possible to point from an ATL transformation rule to any of the target model elements by using the name of a model element. One limitation of this approach is that the end user does not have richer access to the traceability information. For example, searching by type or iterating over all trace links is not supported. [13] propose a method that allows richer run-time access to traceability information by extending the *ASMTransientLink* and *ASMTransientLinkSet* classes. A second limitation of the ATL traceability support is the fact that after the execution of the transformation the traceability information is discarded. In [4] the authors propose a method for persisting traceability information. In this method traceability-specific code is embedded in the transformation code. When a transformation is executed this code generates a traceability model. The traceability-specific code can be added either manually or by the use of a Higher Order Transformation (HOT). Although the proposed solution solves the problem of persistence of traceability information, it creates an overhead problem.

Another transformation language, which provides direct support for traceability, is the Epsilon Transformation Language (ETL) [6]. Similarly to ATL,

ETL uses traceability for resolution of source elements in the target models during a transformation execution. ETL traceability is implicit. When a rule is matched, the ETL engine generates an instance of a *Transformation* class. This instance captures the source element of the rule, a collection of the target elements, and the rule, which was used to create this trace. Once a trace is created it is added to the *TransformationTrace* object, which holds a collection of Traces for a particular transformation. ETL provides the *equivalent()* built-in operation, which uses the generated trace links to automatically resolve source elements to their transformed counterparts. When the *equivalent()* operation is applied on a single source element, it inspects the established transformation trace and invokes the applicable rules (if necessary) to calculate the counterparts of the element in the target model. Epsilon defines a set of model management Ant tasks for orchestration workflows. One of the provided tasks is the *ETL-Task*, whose *exportTransformationTrace* attribute enables developers to export an internal transformation trace to the project context. Finally, ETL supports rule inheritance. When a rule extends another rule, the ETL engine keeps only the sub-rule as part of the trace, ignoring the super-rule.

In Kermet, a traceability framework for facilitating the trace of model transformations is defined [3]. The framework is built atop a language independent traceability metamodel. In Kermet, a model transformation trace is defined as a bipartite graph with source and target nodes. To support transformation chains, Kermet defines every trace as an ordered set of trace steps, each of them representing a single transformation. A trace step can consist of many links, which relate source and target objects. To generate trace information during the execution of a Kermet transformation, traceability-specific code has to be embedded in the transformation code. Finally, the generated trace links can be serialized and subsequently used in further model management tasks.

In the Simple Transformer (SiTra) tool, traceability is inspired by the traceability support of QVT. Tracing consists of the *ITrace* interface, which holds a collection of *TraceInstances*. Each trace represents a mapping between a source and a target model element through a transformation rule. An implementation of the *ITrace* interface provides a number of methods to query the trace collection and return all target instances. The Sitra transformation engine ensures that, for each transformation rule being executed, a trace is recorded.

In the context of QVT, [2] argue that the trace mechanism of the QVT Operational does not capture enough information for common scenarios requiring traceability as input. To enrich the information content of this trace, they propose an alternative metamodel which can be used in conjunction with QVT and it can support richer traceability models.

In the approaches presented so far, there is dedicated support for the creation and usage of traceability. In addition to the aforementioned model transformation languages, there are other languages which do not support directly traceability such as AGG [11], VIATRA [12] and GReAT [1]. Such approaches can support traceability indirectly by creating and manipulating trace links as any other element.

7 Conclusion

We have identified the intent that a single form of traceability should be shared across the family of QVT languages in order to avoid repeated mapping execution and found this to be incompatible with the current specification.

We formulated the intent of traceability as axioms and identified a potential conflict with composition relationships leading to the Child Stealing phenomenon. We have used the axioms to clarify the semantics of transformation extension, mapping refinement, tracing and concurrency in a declarative context.

We have identified significant limitations in the ability to trace reliably in an imperative context and proposed limitations that QVTo could impose to improve tracing integrity.

Acknowledgments Thanks to Adolfo Sanchez-Barbudo Herrera for some helpful comments on an early draft.

References

1. A. Agrawal, G. Karsai, and F. Shi. Graph transformations on domain-specific models. *Journal on Software and Systems Modeling*, 2003.
2. V. Aranega, A. Etien, and J.-L. Dekeyser. Using an alternative trace for qvt. *Electronic Communications of the EASST*, 42, 2011.
3. J.-R. Falleri, M. Huchard, C. Nebut, et al. Towards a traceability framework for model transformations in kermeta. In *ECMDA-TW'06: ECMDA Traceability Workshop*, pages 31–40, 2006.
4. F. Jouault. Loosely coupled traceability for atl. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, volume 91. Citeseer, 2005.
5. F. Jouault and I. Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.
6. D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.
7. B. Liskov. Keynote address - data abstraction and hierarchy. In *ACM SIGPLAN Notices 23 (5)*, pages 17–34. ACM, 1988.
8. B. Meyer. Design by contract. *Technical Report TR-EI-12/CO, Interactive Software Engineering Inc.*, 1986.
9. OMG. Meta object facility (mof) 2.0 query/view/transformation specification, version 1.1 - january 2011. *OMG Document Number: formal/2011-01-01*, 2011.
10. D. Steinberg, F. Budinsky, and E. Paternostro, Marcelo and Merks. Emf: Eclipse modeling framework (2nd edition). *Addison-Wesley*, 2009.
11. G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance*, pages 446–453. Springer, 2004.
12. D. Varró. *VIATRA: visual automated model transformation*. PhD thesis, Thesis, Department of Measurement and Information Systems, University of Technology and Economics, Budapest, 2003.
13. A. Yie and D. Wagelaar. Advanced traceability for atl. In *1st International Workshop on Model Transformation with ATL*, pages 78–87, 2009.