Noname manuscript No.
(will be inserted by the editor)

Least-change Bidirectional Model Transformation with QVT-R and ATL

Nuno Macedo · Alcino Cunha

Received: date / Accepted: date

Abstract QVT Relations (QVT-R) is the standard language proposed by the OMG to specify bidirectional model transformations. Unfortunately, in part due to ambiguities and omissions in the original semantics, acceptance and development of effective tool support has been slow. Recently, the checking semantics of QVT-R has been clarified and formalized. In this article we propose a QVT-R tool that complies to such semantics. Unlike any other existing tool, it also supports meta-models enriched with OCL constraints (thus avoiding returning ill-formed models), and proposes an alternative enforcement semantics that works according to the simple and predictable "principle of least change". The implementation is based on an embedding of both QVT-R transformations and UML class diagrams (annotated with OCL) in Alloy, a lightweight formal specification language with support for automatic model finding via SAT solving. We also show how this technique can be applied to bidirectionalize ATL, a popular (but unidirectional) model transformation language.

Keywords Model transformation · Bidirectional transformation · Least-change principle · QVT-R · ATL · Alloy

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by national funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020532. The first author is also sponsored by FCT grant SFRH/BD/69585/2010.

Nuno Macedo · Alcino Cunha
HASLAB—High Assurance Software Laboratory
INESC TEC & Universidade do Minho
Braga, Portugal
E-mail: {nfmmacedo,alcino}@di.uminho.pt

1 Introduction

Model-Driven Engineering (MDE) is an approach to software development that focuses on models as the primary development entity. In MDE different models may capture different views of the same system (typically different models are used to specify structural and dynamic issues) or may be used at different levels of abstraction (code is obtained by refining platform-independent models to platform-specific ones). All these (possibly overlapping) models should be kept somehow consistent, and changes to one model should be propagated to all the others in a consistent manner. Ideally, specifications of transformations between models should be *bidirectional*, in the sense that a single artifact denotes transformations that can be used in both directions. Moreover, these transformations cannot just map a source to a target model and vice-versa: if some source information is discarded by the transformation, to propagate an update in the target back to a new consistent source, access to the original source model is also required, so that discarded information can be recovered.

To support the MDE approach the Object Management Group (OMG) has launched the Model-Driven Architecture (MDA) initiative, which prescribed the usage of MOF [43] (usually presented as UML class diagrams [41]) and OCL [42] for the specification of (object oriented) models and constraints over them. To specify transformations between models, the OMG proposed the Query/View/Transformation (QVT) standard [40]. While QVT provides three different languages for the specification of transformations, the most relevant to MDE is the *QVT Relations* (QVT-R) language, that allows the specification of a bidirectional transformation by defining a single declarative consistency relation between two (or more) meta-models. Given this specification the transformation can be run in two modes: *checkonly*, to test if two models are consistent according to the specified relation; or *enforce*, that given two models and an execution direction (picking one of them as the target) updates the target model in order to recover consistency. The standard prescribes a "check-before-enforce" semantics, that is, enforce mode cannot modify the target if the models happen to be already consistent according to checking semantics.

Effective tool support for QVT-R has been slow to emerge, which hinders the universal adoption of this standard. In part, this is due to the incomplete and ambiguous semantics defined in [40]. While the checking semantics has recently been clarified and formalized [47, 4, 18], the enforcement semantics still remains largely obscure and even incompatible with other OMG standards, despite some recent efforts to provide a formal specification [5]. Namely, it completely ignores possible OCL constraints over the meta-models, thus allowing updates that can lead to ill-formed target models. Likewise, none of the existing QVT-R model transformation tools supports such constraints, which makes them unusable in many realistic scenarios. Unfortunately, there are other problems that affect them. Some do not even comply to the standard syntax and support only a "QVT-like" language (including not providing both running modes as required by the standard). Others do not support truly non-bijective bidirectional transformations (for example, ignoring the original target model in the enforce mode). Some purposely disregard the intended QVT-R semantics (including checking semantics) and implement a new (still unclear and ambiguous) one. In most cases it is not clear if the supported checking semantics is equivalent to the one formalized in [47,4,18]. And finally, none clarifies the problems and ambiguities in the standard concerning enforcement semantics, and none presents a simple enough alternative for this mode that makes its behavior predictable to the user.

In this article, we propose a QVT-R bidirectional model transformation tool that addresses all these issues. Both the meta-models and transformation specifications may be annotated with OCL, and it supports a large subset of the standard QVT-R language, including execution of both modes independently as prescribed. The main restriction is that recursion must be non-circular (or well-founded), which is satisfied by most of the interesting case-studies. The checking semantics closely follows the one specified in the standard, being equivalent to the one formalized in [47,4,18]. Finally, instead of the ambiguous (and OCL incompatible) enforcement semantics proposed in the standard, our tool follows the clear and predictable *principle of least change* [35], and just returns updated consistent target models that are at a minimal distance from the original. In particular, the "check-before-enforce" policy required by QVT-R is trivially satisfied by this semantics. Our tool supports two different mechanisms to measure the distance between two models: the graph edit distance [51], that just counts insertions and deletions of nodes and edges in the graph that corresponds to a model; and a variation where the user is allowed to parameterize which operations should count as valid edits, by attaching them to the meta-model and specifying their pre- and post-conditions in OCL.

To achieve this, we propose an embedding of both QVT-R transformations and UML class diagrams (annotated with OCL) in Alloy [23], a lightweight formal specification language with support for automatic model finding via SAT solving. Alloy is based on relational logic, which has been shown to be very effective to validate and verify object-oriented models. Its connection with the MDA has also been explored before through tools that translate UML class diagrams annotated with OCL to Alloy [1, 9], on top of which we build our embedding. The proposed technique has been implemented as part of Echo, a tool for the consistent exploration and transformation of models through model finding [33], and has already proved effective in debugging existing transformations, namely helping us unveiling several errors in the well-known object-relational mapping that illustrates the QVT-R specification [40].

Our approach is sufficiently general to be applied to other model transformation languages. To exemplify, we apply our bidirectionalization technique to a significative subset of the *Atlas Transformation Language* (ATL) [24], a widely used, but unidirectional, model transformation language. From the specification of an ATL forward transformation we first infer a consistency relation between source and target meta-models, which then enables us to apply our bidirectionalization engine and automatically obtain a backward transformation that follows the principle of least change.

The present article is an extended version of a previous conference paper [31]. The application of our technique to bidirectionalize ATL is the main new contribution, but in addition to the previous content, this article describes the proposed technique with more detail, how it was deployed in a user-friendly tool, and includes a new case-study and an extensive evaluation to access its effectiveness. Section 2 introduces the QVT-R language, describes the standard checking semantics, presents some of the problems with the enforcement semantics, and proposes and formalizes a simpler alternative based on the principle of least change. Section 3 presents our embedding of UML class diagrams (annotated with OCL) and QVT-R transformations in Alloy. Section 4 explores how the proposed technique can also be used to bidirectionalize ATL. Section 5 describes how it was deployed as part of the Echo framework as an Eclipse IDE plugin for managing of model consistency, while Section 6 presents the evaluation and scalability tests. Finally, Section 7 analyzes related work, while Section 8 draws conclusions and points to future work.

2 QVT Relations

In this section the basic concepts and the semantics of the QVT-R language are introduced. A more detailed presentation can be found in the OMG standard [40].

2.1 Basic Concepts

A QVT-R specification consists of a *transformation* T between a set of meta-models that states under which conditions their conforming models are considered consistent. For the remainder of this article, we will restrict ourselves to transformations between two meta-models for simplicity purposes, although most concepts could be generalized to multi-directional transformations [32]. From T, QVT-R requires the inference of three artifacts: a relation $\boldsymbol{T} \subseteq M \times N$ that tests if two models $m \in M$ and $n \in N$ are consistent and transformations $\overrightarrow{\boldsymbol{T}} : M \times N \to N$ and $\overleftarrow{\boldsymbol{T}} : M \times N \to M$ that propagate changes on a *source* model to a *target* model, restoring consistency between the two. Thus, transformations can be executed in two modes: *checkonly* mode, where the models are simply checked for consistency, denoted as $\boldsymbol{T}\ (m, n)$; and *enforce* mode, where $\overrightarrow{\boldsymbol{T}}$ or $\overleftarrow{\boldsymbol{T}}$ is applied to inconsistent models in order to restore consistency, depending on which of the two models should be updated. Note that both transformations take as extra argument the original opposite model: if models $m \in M$ and $n \in N$ are initially consistent, and m is updated to m', $\overrightarrow{\boldsymbol{T}}$ takes as input both m' and n to produce the new consistent n'. This way the system is able to retrieve from n information discarded by the transformation. This formalization of QVT-R is inspired by the concept of *maintainer* [35], and was first proposed in [46]. Naturally, when the transformations propagate an update the result is expected to be consistent. Formally, the transformation is said to be *correct* if:

$$\forall\ m \in M, n \in N :$$
$$\boldsymbol{T}\ (m, \overrightarrow{\boldsymbol{T}}\ (m, n)) \wedge \boldsymbol{T}\ (\overleftarrow{\boldsymbol{T}}\ (m, n), n)$$

The transformations are also required to follow a "check-before-enforce" policy (also referred to as *hippocraticness* [46]), that can be formalized as follows:

$$\forall\ m \in M, n \in N :$$
$$\boldsymbol{T}\ (m, n) \Rightarrow \overrightarrow{\boldsymbol{T}}\ (m, n) = n \wedge \overleftarrow{\boldsymbol{T}}\ (m, n) = m$$

A QVT-R transformation is defined by a set of *relations.* A relation consists of a *domain pattern* for each meta-model of the transformation, that defines which objects of the model it relates by pattern matching. It also may include *when* and *where* constraints, that act as a kind of pre- and post-conditions for the relation application, respectively. These constraints may contain arbitrary OCL expressions.[In fact, the standard does not allow arbitrary OCL expressions...] The abstract syntax of a relation is the following:

```
[top] relation R {
    [variable declarations]
    domain M a : A { π_M }
    domain N b : B { π_N }
    [when { ψ }]
    [where { φ }]
}
```

In relation R, the domain pattern for meta-model M consists of a *domain variable* a and a template π_M that binds the values of some of its properties (attributes or related associations), which candidate objects of type A must match. Likewise for the domain pattern π_N for meta-model N. To simplify the presentation, the above syntax restricts relations to have exactly one domain variable per meta-model. If the multiplicity of a navigated property R is different from one, pattern templates involving it denote inclusion tests, i.e., a pattern $R = a$ denotes the test $a \in R$. Properties can also be navigated backwards by using the **opposite** keyword. Templates can be complemented with arbitrary OCL constraints. Relations can optionally be marked as **top**, in which case they must hold for all objects of the specified class. Otherwise, they are only tested for particular objects when invoked in **when** or **where** clauses.

2.2 Examples

As a first example, we will define a simplified version of the classic object-relational mapping transformation that illustrates the QVT-R specification [40]. Although simplified, this version still exhibits some of the problems of the original version, which we will describe in the next section. Figure 1 depicts a simplified version of the object (`UML`) and relational (`RDBMS`) meta-models, including signatures of possible edit operations. Figure 2 defines a transformation `uml2rdbms`, whose goal is to map every `persistent Class` in a `Package` to a `Table` in a `Schema` with the same `name`. Each `Table` should contain a `Column` for each `Attribute` (including inherited ones) of the corresponding `Class`. A constraint of the `UML` meta-model that cannot be captured

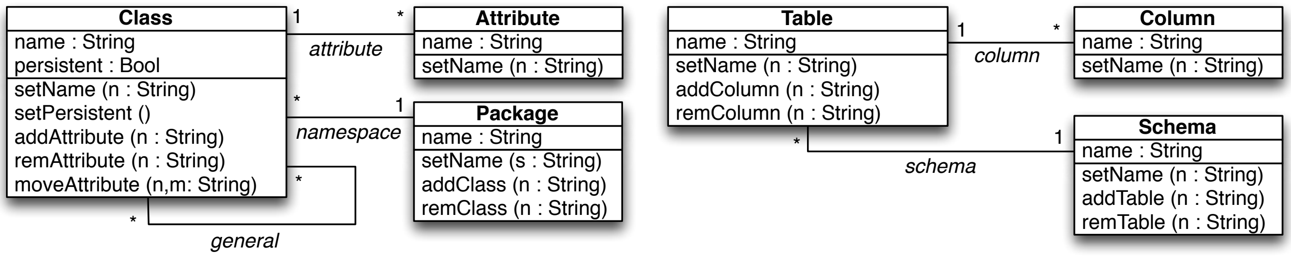


Fig. 1: Class diagrams of the `UML` and `RDBMS` meta-models.

by class diagrams, neither QVT-R key constraints, is the requirement that the association `general` should be acyclic. One must resort to OCL to express it, for example by adding the following invariant to the `UML` meta-model:

```
context Class inv:
    not self.closure(general)->includes(self)
```

The constraint relies on the transitive closure operator, which has recently been introduced to the OCL standard [42, p. 168].

There are two top relations: `P2S` that maps each `Package` to a `Schema` with the same `name`, and `C2T` that maps each `Class` to a `Table` with the same `name`. To ensure that a `Class` is only mapped to a `Table` if the respective `Package` and `Schema` are related, relation `C2T` invokes `P2S` (with concrete domain variables) in the **`when`** clause. For a concrete `Class c` and `Table t`, `C2T` also calls relation `A2C` in the **`where`** clause, that will be responsible to map each `Attribute` in `c` to a `Column` in `t`. `A2C` directly calls `PA2C`, that translates each `Attribute` directly declared in `c` to a `Column` in `t`, and `SA2C`, that recursively calls `A2C` on the `general Class` of `c`, so that each inherited `Attribute` is also translated to a `Column` in `t`.

Another classical bidirectional model transformation example is that of the expansion/collapse of a hierarchical state machine (HSM). In a HSM states may themselves contain sub-states (in which case they are called composite states), as defined by the `HSM` meta-model in Fig. 3. Transitions may exist between sub-states and states outside their owning composite state. Like with the `UML` meta-model, the `HSM` meta-model also requires an additional OCL constraint to avoid circular containment. One advantage of HSMs is abstraction, and a HSM can be collapsed into a non-hierarchical state machine (NHSM) that presents only top-level states, inheriting the incoming and outcoming transitions of their sub-states. The `NHSM` meta-model is similar to `HSM` without the `container` association and the `CompositeState` class, and thus is omitted.

The consistency relation between a HSM and its collapsed view is specified by the `hsm2nhsm` QVT-R transformation in Fig. 4. Top relation `S2S` relates every `State` of a `HSM` with a `NHSM State` with the same `name` as the top-level `State` owning it. The **`where`** clause of top relation `S2S` tests if the `HSM State` is top-level or not: if so, `TS2S` is called, which matches itself to a `NHSM State` with the same `name`; otherwise, `SS2S` is called, which recursively calls `S2S` with its `container State`. Each `Transition` is mapped by the top relation `T2T`, which can be trivially specified by resorting to a **`where`** clause stating that two `Transition`s are related if their `source` and `target State`s are related by `S2S`. Since sub-states in a `HSM` are related to top-states in a `NHSM`, every `Transition` is automatically pushed up to the top-states.

2.3 Checking Semantics

QVT-R's checking semantics assesses if two models are consistent according to the specified transformation. Although the consistency check is by itself important, it is also an essential feature in enforce mode since the latter must "check-before-enforce". The semantics of a relation differs whether it is invoked at the top-level or with concrete domain variables in **`when`** and **`where`** clauses. The specified top-level semantics is directional. As such, from each relation R two consistency relations $R_{\blacktriangleright} : M \times N$ and $R_{\blacktriangleleft} : M \times N$ must be derived, to check if $m : M$ is R-consistent with $n : N$ and if $n : N$ is R-consistent with $m : M$, respectively. The former can be formalized as follows:

$$
\begin{aligned}
&R_{\blacktriangleright}\ (m : M, n : N) \equiv \forall\ xs \mid \psi_{\triangleright} \wedge \pi_M \Rightarrow (\exists\ ys \mid \pi_N \wedge \phi_{\triangleright})\\
&\quad \textbf{where } xs = \mathsf{fv}(\psi \wedge \pi_M) \ \cup\ \{a : A\},\\
&\qquad\qquad ys = (\mathsf{fv}(\pi_N \wedge \phi) \ \cup\ \{b : B\}) - xs
\end{aligned}
$$

Here $\mathsf{fv}(e)$ retrieves the set of free variables from the expression e, so xs denotes the set of variables used in the **`when`** constraint and the source pattern, while ys is the set of variables used exclusively in the **`where`** constraint and in the target pattern. Given a formula ψ, $\psi_{\triangleright}$

```
transformation uml2rdbms (uml:UML,rdb:RDBMS) {
  // PackageToSchema
  top relation P2S {
    n:String;
    domain uml p:Package { name = n };
    domain rdb s:Schema  { name = n };
  }

  // ClassToTable
  top relation C2T {
    n:String;
    domain uml c:Class {
      persistent = true,
      namespace  = p:Package{},
      name       = n
    };
    domain rdb t:Table {
      schema     = s:Schema{},
      name       = n
    };
    when  { P2S(p,s); }
    where { A2C(c,t); }
  }

  // AttributeToColumn
  relation A2C {
    domain uml c:Class {};
    domain rdb t:Table {};
    where { PA2C(c,t) and SA2C(c,t); }
  }

  // PrimitiveAttributeToColumn
  relation PA2C {
    n:String;
    domain uml c:Class {
      attribute  = a:Attribute { name = n }
    };
    domain rdb t:Table {
      column     = cl:Column   { name = n }
    };
  }

  // SuperAttributeToColumn
  relation SA2C {
    domain uml c:Class { general = g:Class {} };
    domain rdb t:Table {};
    where { A2C(g,t); }
  }
}
```

Fig. 2: Simplified version of the `uml2rdbms` QVT-R transformation.

denotes the same formula with all relation invocations replaced by the respective directional version. This semantics is rather straightforward: essentially, for every element $a : A$ that satisfies the **when** condition ψ and matches the π_M domain pattern, there must exist an element $b : B$ that satisfies the **where** condition ϕ and matches the π_N domain pattern. The semantics in the opposite direction is dual. Two models are consistent

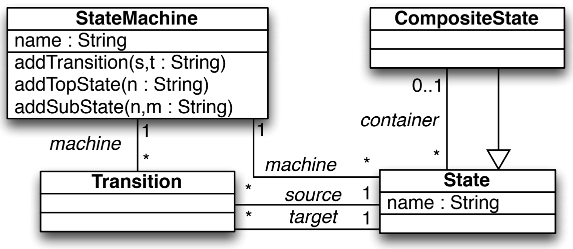


Fig. 3: Class diagram of the `HSM` meta-model.

according to a QVT-R transformation T if they are consistent for all top relations in both directions. Assuming that $\mathcal{R}_T$ is the set of all top level relations of transformation T we have:

$$\boldsymbol{T}\ (m : M, n : N) \equiv \forall\ R : \mathcal{R}_T \mid R_{\blacktriangleright}\ (m, n) \wedge R_{\blacktriangleleft}\ (m, n)$$

The QVT-R standard [40] defines rather precisely the top-level semantics, but is omissive about the semantics of relations invoked with concrete domain variables. Recent works on the formalization of QVT-R check semantics [47,4,18] clarify that it is essentially the same as the top-level—still directional, but defined over specific objects by fixing the domain variables. As such, from each relation R with domain variables of type A and B, two consistency relations $R_{\triangleright} : M \times N \times A \times B$ and $R_{\triangleleft} : M \times N \times A \times B$ are inferred, to check if two concrete objects a and b (belonging to models $m : M$ and $n : N$, respectively) are consistent: [Just a comment... not to be taken too seriously, but probably it would be more precise if in quantifications we indexed the types by the model, to mean it is a quantification over the elements of that type in that particular model. This would actually make the transition to the local state idiom in Alloy more smooth.]

$$\begin{aligned}
&R_{\triangleright}\ (m : M, n : N, a : A, b : B) \equiv \\
&\quad \forall\ xs \mid \psi_{\triangleright} \wedge \pi_M \Rightarrow (\exists\ ys \mid \pi_N \wedge \phi_{\triangleright}) \\
&\qquad \textbf{where}\ xs = \mathsf{fv}(\psi \wedge \pi_M), \\
&\qquad\qquad\quad ys = \mathsf{fv}(\pi_N \wedge \phi) - xs
\end{aligned}$$

Although it may be tempting (and probably more intuitive) to define $R_{\blacktriangleright}$ in terms of $R_{\triangleright}$, that is $R_{\blacktriangleright}\ (m, n) \equiv \forall\ a : A \mid \exists\ b : B \mid R_{\triangleright}\ (m, n, a, b)$, this definition is not semantically equivalent to the one presented above, as already discussed in [4]. For instance, consider the semantics (in the direction of `UML`) of relation `PA2C` from the `uml2rdbms` transformation:

$$\begin{aligned}
&\texttt{PA2C}_{\blacktriangleleft}\ (uml : \texttt{UML}, rdb : \texttt{RDBMS}) \equiv \\
&\quad \forall\ t : \texttt{Table}, cl : \texttt{Column}, n : \texttt{String} \mid \\
&\qquad cl \in t.\texttt{column} \wedge cl.\texttt{name} = n \Rightarrow \\
&\qquad\quad \exists\ c : \texttt{Class}, a : \texttt{Attribute} \mid
\end{aligned}$$

```
transformation hsm2nhsm (hsm : HSM, nhm : NHSM) {
  // StateMachineToStateMachine
  top relation M2M {
    n:String;
    domain hsm s:StateMachine { name = n };
    domain nhm t:StateMachine { name = n };
  }

  // StateToState
  top relation S2S {
    domain hsm s:State {
      machine = sm:StateMachine{} };
    domain nhm t:State {
      machine = tm:StateMachine{} };
    when  { M2M(sm,tm); }
    where {
      if s.container->isEmpty() then TS2S(s,t)
      else SS2S(s,t) endif;
    }
  }

  // TopState2State
  relation TS2S {
    n: String;
    domain hsm s:State { name = n };
    domain nhm t:State { name = n };
  }

  // SubState2State
  relation SS2S {
    domain hsm s:State {};
    domain nhm t:State {};
    where { S2S(s.container,t); }
  }

  // TransitionToTransition
  top relation T2T {
    domain hsm ht:Transition {
      target = htt:State{},
      source = hts:State{}
    };
    domain nhm nt:Transition {
      target = ntt:State{},
      source = nts:State{}
    };
    where { S2S(hts,nts) and S2S(htt,ntt); }
  }
}
```

Fig. 4: The `hsm2nhsm` QVT-R transformation.

$$
\begin{aligned}
&\qquad\quad a \in c.\mathsf{attribute} \wedge a.\mathsf{name} = n\\
&\mathsf{PA2C}_{\lhd}\ (uml : \mathsf{UML}, rdb : \mathsf{RDBMS}, c : \mathsf{Class}, t : \mathsf{Table}) \equiv\\
&\quad \forall\ cl : \mathsf{Column}, n : \mathsf{String} \mid\\
&\qquad cl \in t.\mathsf{column} \wedge cl.\mathsf{name} = n \Rightarrow\\
&\qquad\quad \exists\ a : \mathsf{Attribute} \mid\\
&\qquad\qquad a \in c.\mathsf{attribute} \wedge a.\mathsf{name} = n
\end{aligned}
$$

Consider a simple `UML` model where a `Class` a with an `Attribute` x extends a `Class` b with an `Attribute` y. Consider also a `RDBMS` model with a single `Table` a containing a `Column` x and a `Column` y. While $\mathsf{PA2C}_{\blacktriangleleft}$ holds for this pair of models, $\mathsf{PA2C}_{\lhd}$ returns false for every pair of `Class` and `Table`. Of course, there are cases where the two semantics are equivalent. For instance, `C2T` could be defined as a non-top relation and be called from the **`where`** clause of `P2S`. The behavior is equivalent because the only free variable (n) is bound to a unitary attribute.

Due to this asymmetry and the directionality of the semantics, QVT-R transformations may not have the expected behavior. In particular, `uml2rdbms` as defined in the standard does not have a bidirectional semantics, in the sense that the only pairs of consistent and valid finite models are ones where all classes are non-persistent and there are no tables. To see why this happens, consider the relations `A2C` and `SA2C` when checked in the direction of `UML`. These relations call each other recursively, and their non top-level semantics is:

$$
\begin{aligned}
&\mathsf{A2C}_{\lhd}\ (uml : \mathsf{UML}, rdb : \mathsf{RDBMS}, c : \mathsf{Class}, t : \mathsf{Table}) \equiv\\
&\quad \mathsf{PA2C}_{\lhd}\ (uml, rdb, c, t) \wedge \mathsf{SA2C}_{\lhd}\ (uml, rdb, c, t)\\
&\mathsf{SA2C}_{\lhd}\ (uml : \mathsf{UML}, rdb : \mathsf{RDBMS}, c : \mathsf{Class}, t : \mathsf{Table}) \equiv\\
&\quad \exists\ g : \mathsf{Class} \mid g \in c.\mathsf{general} \wedge \mathsf{A2C}_{\lhd}\ (uml, rdb, g, t)
\end{aligned}
$$

If the transformation takes into account the OCL constraint requiring `general` to be acyclic, the predicate $\mathsf{A2C}_{\lhd}\ (uml, rdb, c, t)$ never holds in a finite model, since c will be required to have an infinite ascending chain of `general` objects. This is due to the under-restrictive `SA2C` domain pattern in the `RDBMS` side (empty in this case), that requires every `Table` to have a matching `Class` with a `general`, which, due to recursion, is also required to have a `general`, and so on. This is but one of the problems that occur in the original specification of this transformation, and is another example of the ambiguities that prevail in the QVT standard [40]: while it requires consistency to be checked in both directions, the case-study used to illustrate it was clearly not developed with bidirectionality in mind. Note that checking consistency only in the direction of `RDBMS` does not suffice, since, for example, it will not prevent spurious tables to appear in the target schema.

Concerning recursion we can distinguish two situations: one is well-founded recursion, where the call graph of the transformation contains a loop, but in any evaluation it is traversed only finitely many times; another is cyclic (or infinite) recursion, where such a loop may actually be traversed infinitely many times (e.g., when a relation directly or indirectly calls itself with the same arguments). The semantics of well-founded recursion is not problematic, but the standard is omissive about what should happen when infinite recursion occurs. A possible interpretation is that it should not be allowed, although in general it is undecidable to detect if that is the case. Similarly to some QVT-R formalizations [47,18], the embedding presented in this article is not well-defined when infinite recursion occurs.

Recently, a formal semantics of QVT-R was proposed [4] that is well-defined even in presence of infinite recursion, by resorting to modal mu calculus. To see why taking OCL constraints into account is fundamental, a transformation conforming to this semantics, but that ignores the requirement that `general` is acyclic, would consider an (ill-formed) `UML` model with a single persistent `Class` a that generalizes itself consistent with a `RDBMS` model with a `Table` a.

To prevent the problem in the `uml2rdbms` transformation described above, one could tag each `Column` with the path to the particular `general` they originated from, and then refine the `RDBMS` domain pattern to prevent problematic recursive calls. A simpler alternative is to resort to the transitive closure operation, and map at once every declared or inherited `Attribute` of a given `Class` to a `Column` of the respective `Table`. In this new version of `uml2rdbms` (that will be considered in the remainder of the article), `A2C`, `PA2C` and `SA2C` are replaced by the following alternative definition of `A2C`:

```
relation A2C {
  cn:String; a:Attribute; g:Class;
  domain uml c:Class {} {
    (c->closure(general)->includes(g) or g = c) and
    g.attributes->includes(a) and a.name = cn
  };
  domain rdb t:Table {
    column = cl:Column { name = cn }
  };
}
```

The additional OCL constraint in the `UML` domain pattern acts as a pre-condition when applying the transformation in the direction of `RDBMS`, and as a post-condition in the other direction. As such, it could not be specified in the **`when`** clause, since it would act as (an undesired) pre-condition for both scenarios.

Unlike `uml2rdbms`, the recursive version of `hsm2nhsm` does produce the intended behavior. The reason is that, while attributes in `uml2rdbms` may give origin to columns in multiple tables, `HSM` transitions in `hsm2nhsm` give rise to a single `NHSM` transition. As a consequence, unlike `A2C` that must be defined over classes, `T2T` can be defined directly over transitions. These particularities are difficult to grasp at design time, thus effective tool support for QVT-R is essential for the development consistency relations that embody the intentions of the user.

2.4 Enforcement Semantics

Despite showing many ambiguities and omissions, we believe that, due to the reasons presented next, the enforcement semantics intended in the standard for this mode is quite undesirable. Instead, we propose an alternative that is easy to formalize, more flexible, and more predictable to the end-user.

In the QVT-R standard, update propagation is required to be deterministic. This is a desirable property, since it makes its behavior more predictable. However, to ensure determinism, every transformation is required to follow very stringent syntactic rules that reduce update translation to a trivial imperative procedure. Namely, it should be possible to order all constraints in a relation (except for the target domain pattern), such that the value of every free variable is fixed by a previous constraint. Although not clarified in the standard, this means that every relation invoked in **`when`** and **`where`** constraints is either invoked with previously bound variables, or required to also be deterministic, even if the intention was to only make update propagation deterministic. For example, in transformation `uml2rdbms`, update propagation in the `RDBMS` direction will only be deterministic for relation `C2T` if at most one s is consistent with p according to relation `P2S` (note that s is still free in the **`when`** clause). In this particular example that happens to be true, but in general such determinism is undesirable since it forces relations to be one-to-one mappings, limiting the expressiveness of the language. Moreover, it defeats the purpose of a declarative transformation language, since one is forced to think in terms of imperative execution and write more verbose transformations. For example, our simpler version of `A2C` using transitive closure would not be allowed, since the value of g is not known a priori when enforcing consistency in the direction of `UML`.

Another problem is the predictability of update propagation. Being deterministic is just part of the story—it should be clear to the user why some particular element was chosen to be updated instead of another. The only mechanism proposed by QVT-R to control updatability are *keys*. For example, one could add the statement **`key`** `Table (name, schema);` to the running example to assert that each `Table` is uniquely identified by the pair of properties `name` and `schema`. If an update is required on a `Table` to restore consistency (for example, when an `Attribute` is added to a `Class`), such key is used to find a matching `Table`. When found, an update is performed, otherwise a new `Table` is created. This works well when all domains involved in relations have natural keys, which again points to one-to-one mappings only, but fails if such keys do not exist. In those cases, the standard prescribes that update propagation should always be made by means of creation of new elements, even if sometimes a simple update to an existing element would suffice. Since creation requires defaults for mandatory (multiplicity one) properties, this would result in models with little resemblance with the original (which would basically be discarded).

Our alternative enforcement semantics is based on the *principle of least change*, first proposed in the context of *maintainers* [35], and that promotes predictability by requiring updates to be as small as possible. QVT-R "check-before-enforce" policy is just a particular case of this more general principle. Let $\Delta_M : M \times M \to \mathbb{N}$ be an operation that computes the update distance between M models. Then, the principle of least change states that the models returned by the transformations $\overrightarrow{\boldsymbol{T}}$ and $\overleftarrow{\boldsymbol{T}}$ are just the consistent models closest to the original. Formally, we have:

$$\begin{aligned}
&\forall\ m \in M, n, n' \in N : \boldsymbol{T}\ (m, n') \Rightarrow \\
&\quad \Delta_N\ (\overrightarrow{\boldsymbol{T}}\ (m, n), n)\ \leqslant\ \Delta_N\ (n', n) \\
&\forall\ m, m' \in M, n \in N : \boldsymbol{T}\ (m', n) \Rightarrow \\
&\quad \Delta_M\ (\overleftarrow{\boldsymbol{T}}\ (m, n), m)\ \leqslant\ \Delta_M\ (m', m)
\end{aligned}$$

Assuming that the distance is only null when the model is unchanged (i.e., $\Delta\ (n, n') = 0 \equiv n = n'$), it is trivial to show that these laws reduce to hippocraticness when the models m and n are already consistent. Note, that this principle by itself does not ensure determinism, although it substantially reduces the set of possible results. If among the returned models the user wishes to favor a particular subset, keys or OCL constraints can be added to the meta-model to further guide the transformation engine.

We propose two different techniques to measure the update distance between models. In the first one, models are interpreted as graphs and the *graph edit distance* (GED) [51] is measured. GED measures the distance between two graphs as the number of node and edge insertions and deletions needed to obtain one from the other. Concretely, graph nodes denote model elements and literal values (i.e., primitive type values or enumeration literals), while edges denote links between model elements or attributes between model elements and literal values. GED counts changes in this graph representation, with the exception of literal values, which are considered external to particular model instances and thus do not affect model distance. This is a meta-model independent metric that is automatically inferred by our tool for any meta-model provided by the user.

The simple definition for distance provided by GED assumes a fixed repertoire of edit operations which may not be desirable. In particular, there is no control over the "cost" of complex operations. For example, changing the `name` of a `Class` will have a cost of 2, since it requires deleting the current `name` edge and inserting a new one, while adding a new `Attribute` to a `Class` will cost 3, since it requires creating a new `Attribute`, setting its `name`, and adding it to the `Class`. One may wish both these operations to be atomic edits and have the same unitary cost. Also, one may wish to allow only particular edits in order to control non-determinism of enforcement runs.

To address such limitations, we propose as an alternative measure an *operation-based distance* (OBD), that allows the user to control the range of valid repairs by specifying in the meta-model which edit operations can be applied to update the model. These are specified using pre- and post-conditions defined in (a subset of) OCL. For the purposes of our running example, we assume the existence of the edit operations whose interfaces are defined in Fig. 1. The following is an OCL specification of the operation `setName` from `Class`:

```
context Class::setName(n : String)
  post name
    self.name = n;
  post frame_class_name
    Class.allInstances()->forAll(c |
      c.name@pre = c.name or c = self)
  modifies Class::name
```

In this case, Δ will be the length of the edit operation sequence (built over the user-defined operations) required to achieve the new model. Enforcing the principle of least change entails minimizing this sequence between the original and the updated models. While OBD allows the assignment of lower costs to complex updates (simply create an operation that composes smaller operations), assigning higher costs to simple operations is not as straight-forward as they may not be decomposable. This would require customizable operation costs which is left as future work.

A source of ambiguity in OCL concerns frame conditions. Assuming that everything that is not mentioned in the post-condition is not changed is generally a reasonable assumption, but this is not trivial to infer from declarative specifications. Given the lack of OCL statements focusing on frame conditions, we introduce "*modifies*" clauses, through which the user must explicitly specify which elements of the model may be modified by the operation—the remainder are assumed to remain unchanged. This mechanism is similar to those introduced by behavioral interface specification languages, like the *Java Modeling Language* (JML) [29]. In the previous example, the **`modifies`** keyword states that only the attribute `name` in `Class` is modified by operation `setName`.

While our semantics, following the constraint maintainers framework and the QVT-R standard, was developed in a bidirectional transformation scenario (in the sense that consistency is restored by updating a single model), it is worth noting that it could trivially be adapted to the general synchronization scenario where both models can be updated simultaneously: resorting to the same consistency relation, enforcement runs would try to minimize the distance of both models to the original ones, rather than just one. Given a synchronization procedure $\overleftrightarrow{\boldsymbol{T}} : M \times N \to M \times N$ and a distance metric over pairs of models $\Delta_{M\times N} : (M \times N) \times (M \times N) \to \mathbb{N}$, typically

$$\Delta_{M\times N}\ (m,n)\ (m',n') = \Delta_M\ (m,m') + \Delta_N\ (n,n'),$$

the least-change principle would be formalized as:

$$\forall\ m,m' \in M, n,n' \in N : \boldsymbol{T}\ (m',n') \Rightarrow \\ \Delta_{M\times N}\ (\overleftrightarrow{\boldsymbol{T}}\ (m,n),(m,n))\ \leqslant\ \Delta_{M\times N}\ ((m',n'),(m,n))$$

This is related to the multi-directional scenario, where the user may wish to update multiple models in order to restore consistency, to which our technique can also be generalized [32]. Here, the system tries to minimize the distance between the set of original and target models that the user chose as targets of the enforcement run.

3 Embedding QVT-R in Alloy

In this section we present how the semantics proposed in the previous section can be operationalized by an embedding in Alloy. To keep the paper self-contained, a brief introduction to Alloy is presented, focusing on the concepts deemed essential to understand our embedding; for a deeper exposition the reader is redirected to [23]. The reader not interested in the technical details of the embedding can skip over to Section 4.

3.1 A Brief Introduction to Alloy

Alloy is a lightweight formal specification language that, supported by the Alloy Analyzer, provides bounded model checking and model finding functionalities through an embedding in off-the-self SAT solvers. Alloy is a rich and flexible language; in this section we focus only on concepts deemed essential for the scope of this article.

An Alloy specification is developed in *modules*, that consist of *paragraphs*: signature declarations, constraints and commands. A *signature* declaration introduces a set of elements sharing a similar structure and properties. In Alloy such elements are uninterpreted, immutable and indivisible, and are thus denoted *atoms*. A signature declaration may also introduce *fields*, i.e. relations that connect its atoms to those of other (or the same) signatures. These are represented as sets of tuples of atoms in instances. Alloy is not restricted to binary relations, and it is not uncommon to have fields that relate three or more signatures. A signature that **extends** other signatures inherits their fields. It can also be contained **in** another signature, in which case it is simply a subset of the parent signature.

Signatures may be annotated with *multiplicity* keywords to restrict their cardinality, namely **some** (at least some elements), **lone** (at most one element), and **one** (exactly one element). The range signature in a field declaration can also be annotated with such multiplicities, to restrict the number of atoms that can be connected to each atom of the source signature. If that number is arbitrary, the special multiplicity keyword **set** should be used.

Facts specify properties that must hold in every instance. These may call *functions* and *predicates*, that are essentially containers for reusable expressions. *Commands* are used to perform particular analyses, by invoking the underlying solver. *Run* commands try to find instances for which the specified properties hold, while *check* commands try to find counter-examples that refute them. Commands can be parametrized by scopes for the declared signatures, thus bounding the search-space for the solver. If no scope is specified a default of 3 is assumed.

Figure 5 depicts a possible (incomplete) specification of the UML class diagram meta-model using Alloy. Signatures `Package`, `Class` and `Attribute` declare the corresponding classes and introduce (binary) fields to represent the classes' attributes and associations. Alloy does not have a primitive boolean type, so boolean attributes are usually represented by subset signatures containing the elements that have the attribute set to true. This is the case of the `persistent` attribute of `Class`, here represented by the `Persistent` subset signature. The run command instructs the analyzer to search for instances conforming to the `acyclic` predicate, setting a specific scope for each of the signatures.

Formulas in Alloy are defined in relational logic, an extension of first-order logic with relational and closure operators. Everything in Alloy is a *relation*, i.e. a set of tuples of atoms (with uniform arity). Signatures are unary relations (sets) containing the respective atoms and scalar values (including quantified variables) are just singleton sets. This uniformity of concepts leads to a very simple semantics. The relational logic operators also favor a navigational style of specification that is appealing to software engineers, as it resembles object-oriented languages.

The key operator in Alloy is the *dot join* composition that allows the navigation through fields (and relational expressions in general). For example, if `c` is a `Class`, `c.name` denotes its name (a scalar) and `c.general` accesses its super-class (a set containing at most one `Class`). Besides composition, relational expressions can

```
module UML

sig Package {
  name : one String
}
sig Attribute {
  name : one String
}
sig Class {
  attribute : set Attribute,
  general   : lone Class,
  namespace : one Package,
  name      : one String
}
sig Persistent in Class {}

pred acyclic {
  all self:Class | self not in self.^general
}

run { acyclic }
  for 3 Class, 3 Attribute, 1 Package, 3 String
```

Fig. 5: A (static) specification of `UML` in Alloy.

also be built using the union (`+`), intersection (`&`), difference (`-`), and cartesian product (`->`) operators. In particular, singleton tuples can be defined by taking the cartesian product of two (or more) scalars. Relations can also have their domain restricted to a given set (`<:`) and likewise for the range (`:>`). For example, `Persistent <: name` is the binary relation that associates persistent classes with the respective names. Binary relational expressions can also be reversed (~), extended with the transitive closure (`^`), or with the reflexive transitive closure (∗). For example, in the `acyclic` predicate, expression `self.^general` retrieves all the super-classes of `self`. Relational expressions may also be created by set comprehension. Finally, there are some primitive relations pre-defined in Alloy: **`univ`** denotes the *universe*, i.e. the set of all tuples, **`none`** denotes the empty set, and **`iden`** the binary identity relation over the universe.

Alloy has limited support for integers: the pre-defined **`Int`** signature contains all available integers. In commands, the scope of **`Int`** determines the available number of bits to represent them (in two's complement notation). Integers can be added and subtracted with the functions **`plus`** and **`minus`**, respectively. The default semantics for integer operations is wrap around: for example, if the scope for **`Int`** is 3, **`plus`**`[3,1]` is `-4`. Every relation expression can have its cardinality determined with the `#` operator.

Atomic formulas are built from relational expressions using inclusion (**`in`**), equality (`=`), or cardinality checks (besides **`lone`**, **`some`**, and **`one`**, keyword **`no`** can also be used to check if a relational expression is empty). Formulas can be combined with conjunction (`&&`), disjunction (`||`), implication (`=>`), possibly associated with an **`else`** formula, equivalence (`<=>`), and negation (**`not`**). Besides the universal (**`all`**) and existential (**`some`**) quantifiers, Alloy also supports **`lone`** (property holds for at most one atom), **`one`** (property holds for exactly one atom), and **`no`** (property holds for no atom) quantifiers. In the `acyclic` predicate, as expected, the formula quantifies over all atoms of signature `Class` and tests if the inheritance chain is acyclic.

3.2 Meta-models Annotated with OCL

The models upon which our transformations are defined consist of UML class diagrams annotated with OCL constraints. Some translations have been proposed to embed such models in Alloy, namely [1,9]. Our embedding will be based on the translation proposed in [9], since, unlike other proposals, it covers an expressive OCL subset that includes closure and operation specification via pre- and post-conditions. Here, we will just briefly present this translation.

Classes, their attributes, and related associations can be directly translated to signatures and fields in Alloy. Likewise for the inheritance relationship, that Alloy also supports. The main difference between the embedding from the previous section is that, since Alloy instances are built from immutable atoms, the transformation resorts to the well-known *local state idiom* [23] to capture updates to a given model. This means that a special signature will be introduced to represent each meta-model, whose atoms will denote different models (or evolutions of a given model). To each field (representing an association or an attribute) an extra column of this type is added, to allow its value to change in different models. The translation proposed in [9] is also extended to allow classes to have different elements in different models: for each class a special binary field (with the same name) will capture the objects of that class that exist in each model, to which we will refer as the signature's *state field*. Boolean attributes are encoded similarly: a binary field captures which objects have the attribute set to `true` in each model. For example, class `Class` of our `UML` meta-model is translated to the following signature declaration.

```
sig UML {}
sig Class {
  class      : set UML,
  attribute  : Attribute -> UML,
  general    : Class -> UML,
  namespace  : Package -> UML,
  name       : String -> UML,
  persistent : set UML
}
```

The binary state field `class` captures the `Class` objects that exist in each `UML` model. The remaining fields model the respective `Class` associations and attributes. With the relational composition operator we can access the values of these fields for a given `UML` model `m`. For example, `class.m` is the set of `Class` objects that exist in model `m`, `general.m` is a binary relation that maps each `Class` to its `general` in model `m`, and `persistent.m` is the set of `Class` objects that have the attribute `persistent` set to `true` in model `m`.

Constraints must also be generated to ensure the correct multiplicities, and that fields only relate atoms existing in the same model (inclusion dependencies). For example, fact

```
all m:UML | namespace.m in class.m -> one package.m
```

is generated to capture the cardinality constraints of relation `namespace`, and to force it, for each `UML` model `m`, to be a subset of the cartesian product between `class.m` and `package.m` (respectively, the sets of `Class` and `Package` elements of model `m`). Constraints that guarantee the integrity of the class hierarchy are also inserted, for instance, in the `HSM` meta-model, fact

```
all m:HSM | compositestate.m in state.m
```

would ensure that if a `CompositeState` exists in model `m`, it is also registered as a `State` in that model. OCL invariants in a given context are also automatically translated to Alloy facts, resulting in universal quantifications over the respective signature state fields, following the technique previously developed in [9]. Table 1 summarizes the currently supported operations from the OCL standard library [42] (operation **`oclIsNew`** may only be used in controlled contexts as explained in Section 3.4).For example, the OCL invariant stating that association `general` is acyclic is translated to Alloy as

```
all m:UML, self:class.m |
  self not in self.^(general.m)
```

Here, `^(general.m)` is the transitive closure of field `general` projected over `m`.

The QVT standard extends the OCL language with the insertion of the **`opposite`** keyword that allows the navigation of associations in the opposite direction, which can be directly translated to Alloy using the converse operator $\sim$.

3.3 QVT-R Transformations

Top relations $R_\blacktriangleright$ and $R_\blacktriangleleft$ are specified by predicates parameterized by the model instances. The definition of all these predicates follows closely the formalization

family	operations
base	=, <>, **oclIsNew**, **oclIsKindOf**, **oclType**, **oclAsType**, **allInstances**
integer	+, -, >, <, <=, >=
boolean	**and**, **or**, **not**, **implies**, **true**, **false**
set	**size**, **includes**, **includesAll**, **excludes**, **excludesAll**, **isEmpty**, **notEmpty**, **union**, -, **intersection**, **including**, **excluding**, **asSet**
iterators	**exists**, **forAll**, **one**, **any**, **collect**, **select**, **reject**, **closure**
QVT	**opposite**

Table 1: Supported OCL operations.

in Section 2.3. In particular, auxiliary predicates are used to specify the **`when`** and **`where`** clauses, and the domain patterns of each relation. For example, back to `uml2rdbms`, Fig. 6 presents the result of embedding $\mathtt{C2T}_\blacktriangleright$ in Alloy as the predicate `Top_C2T_RDBMS`, as well as the necessary auxiliary predicates. Note how, in the specification of $\mathtt{C2T}_\blacktriangleright$, quantifications are restricted to range over atoms existing in the respective models.

For each relation R we also declare two Alloy predicates to specify $R_\triangleright$ and $R_\triangleleft$. For example, in Fig. 6 the omitted predicates `P2S_RDBMS` and `A2C_RDBMS` specify $\mathtt{P2S}_\triangleright$ and $\mathtt{A2C}_\triangleright$, respectively. Besides the respective domain elements, these are also parameterized by the models they are being applied to. Since in Alloy predicates cannot call each other recursively, predicates $R_\triangleright$ and $R_\triangleleft$ are defined in terms of auxiliary relations over the model state signatures, specified by comprehension. For instance, the following recursive predicates, that would arise from a direct encoding of $\mathtt{S2S}_\triangleright$ and $\mathtt{SS2S}_\triangleright$ in `hsm2nhsm`, are invalid in Alloy.

```
pred S2S_NHSM [hsm:HSM,nhm:NHSM,s:State,t:State] {
    no s.container => TS2S_NHSM[hsm,nhm,s,t]
                else SS2S_NHSM[hsm,nhm,s,t]
}

pred SS2S_NHSM [hsm:HSM,nhm:NHSM,s:State,t:State] {
    S2S_NHSM[hsm,nhm,s.container,t]
}
```

Instead, we declare auxiliary relations `S2S_NHSM'` and `SS2S_NHSM'` with types `HSM->NHSM->State->State` and `HSM->NHSM->State->State`, respectively, and axiomatize their value using set comprehension as follows:

```
fact {
  S2S_NHSM'  = { hsm:HSM,nhm:NHSM,s:State,t:State |
    no s.container => hsm->nhm->s->t in TS2S_NHSM'
                else hsm->nhm->s->t in SS2S_NHSM' }

  SS2S_NHSM' = { hsm:HSM,nhm:NHSM,s:State,t:State |
    hsm->nhm->s.container->t in S2S_NHSM' }
}
```

Note how predicate invocation is replaced by membership check: for example, instead of the predicate

```
pred When_C2T_RDBMS [uml:UML,rdb:RDBMS,p:Package,s:Schema] {
    P2S_RDBMS[uml,rdb,p,s]
}

pred Where_A2C_RDBMS [uml:UML,rdb:RDBMS,c:Class,t:Table] {
    A2C_RDBMS[uml,rdb,c,t]
}

pred Pattern_C2T_UML [uml:UML,c:Class,n:String,p:Package] {
    n in c.name.uml && c in persistent.uml && p in c.namespace.uml
}

pred Pattern_C2T_RDBMS [rdb:RDBMS,t:Table,n:String,s:Schema] {
    s in t.schema.rdb && n in t.name.rdb
}

pred Top_C2T_RDBMS [uml:UML,rdb:RDBMS] {
    all c:class.uml, n:String, p:package.uml, s:schema.rdb |
        When_C2T_RDBMS[uml,rdb,p,s] && Pattern_C2T_UML[uml,c,n,p] =>
            some t:table.rdb | Pattern_C2T_RDBMS[rdb,t,n,s] && Where_A2C_RDBMS[uml,rdb,c,t]
}
```

Fig. 6: Alloy specification of $\mathsf{C2T}_{\blacktriangleright}$.

call TS2S_NHSM[hsm,nhm,s,t] we check that the tuple hsm->nhs->s->t is included in relation TS2S_NHSM'. By resorting to these, the predicate encodings $\mathsf{S2S}_{\triangleright}$ and $\mathsf{SS2S}_{\triangleright}$ can now be redefined simply as

```
pred S2S_NHSM [hsm:HSM,nhm:NHSM,s:State,t:State] {
  hsm->nhm->s->t in S2S_NHSM'
}

pred SS2S_NHSM [hsm:HSM,nhm:NHSM,s:State,t:State] {
  hsm->nhm->s->t in SS2S_NHSM'
}
```

As discussed in Section 2.3, this embedding will not be well-behaved in presence of cyclic recursion.

The checking semantics of the transformation is represented by a predicate that checks all top relations in both directions. In the uml2rdmbs example we have:

```
pred uml2rdbms [uml:UML,rdb:RDBMS]{
  Top_P2S_RDBMS[uml,rdb] && Top_P2S_UML[uml,rdb] &&
  Top_C2T_RDBMS[uml,rdb] && Top_C2T_UML[uml,rdb]
}
```

Regarding enforcement semantics, to implement the principle of least change as described in Section 2.4, we require the measurement of the update distance between two models. The first proposed metric is GED, that interprets models as graphs and measures the distance as the number of node and edge insertions and deletions needed to obtain one graph from the other. Note that an Alloy instance is isomorphic to a labelled graph whose nodes are the atoms, and edges tuples in fields (technically to hypergraphs, since fields are n-ary). With this mechanism, Δ_{UML} can be computed as follows:

```
fun Delta_UML [m,m':UML] : Int {
  #((class.m-class.m')+(class.m'-class.m)).plus[
  #((name.m-name.m')+(name.m'-name.m)).plus[
  ... // symmetric difference of remainder fields
  ]]
}
```

Assuming m' represents an updated version of m, this function sums up, for every signature and field, the size of their symmetric difference in both models. To avoid Alloy's standard wrap around semantics for integers, model finding is executed with the option *Forbid Overflow* set [36].

Regarding OBD, the edit operations, specified by the user in OCL using pre- and post-conditions, are automatically converted to Alloy using the translation procedure defined in [9]. Essentially, each operation will originate an Alloy predicate that specifies when it can hold between two models. The resulting Alloy predicate takes as arguments the pre- and post-states of the affected model (with the post-state being denoted by a primed variable), the receiver element of the edit operation (denoted by argument self of the appropriate context class), as well as the stated operation parameters. For example, the result of translating setName to Alloy is the following:

```
pred setName[self:Class,n:String,m,m':UML] {
  self.(name.m') = n;
  all c:class.m' |
    c.(name.m) = c.(name.m') or c = self
  // frame conditions inferred from modifies
  class.m' = class.m
  attribute.m' = attribute.m
  general.m' = general.m
  namespace.m' = namespace.m
  ... // remaining frame conditions
 }
```

The body of the predicate consists of the translation of the pre- and post-conditions from the OCL specification. Pre-conditions (if any) are evaluated over the pre-state of the `UML` model `m`, while post-conditions refer to the respective post-model `m'`, except in the case of operations and properties marked by the tag **`@pre`** which are still evaluated in the pre-state. Frame conditions for all classes and associations not included in the **`modifies`** clause are also automatically inferred.

Given the specifications of operations, we constrain models to form a sequence, where each step corresponds to the application of an edit operation:

```
open util/ordering[UML]
fact {
 all m:UML, m':m.next | {
  some c:class.m,n:String | setName[c,n,m,m'] or
  some c:class.m,n:String | addAttribute[c,n,m,m'] or
  ...  // remaining operation predicates
 }
}
```

The Alloy module `ordering` imposes a total order on all atoms of the given signature (in this case, `UML`), and declares a binary relation `next` that captures such order. The presented fact restricts the possible values of `next`, by requiring each state `m` and subsequent state `m.next` to be related by one of the specified operations.

In this case, Δ_{UML} will be the number of models (intermediate steps) required to achieve a consistent target, which, as we will see next, will be determined by the scope of the signature denoting the respective meta-model, `UML` in this case.

3.4 Executing the Semantics

Executing the transformation in checkonly mode is fairly simple: we just need to check the consistency predicate for a pair of concrete models. To represent a concrete model, since Alloy has no specific constructs to denote model instances, we use singleton signatures to denote specific objects and facts to fix the interpretation of fields. For example, a `UML` model `M` with `Class A` and `Class B`, with no `Attribute` elements, in a single `Package P`, where `A` is `persistent` and extends the non-persistent `B`, can be specified as follows:

```
one sig M extends UML {}
one sig P extends Package {}
one sig A,B extends Class {}
fact {
    class.M       = A + B &&
    package.M     = P &&
    namespace.M   = A->P + B->P &&
    general.M     = A->B &&
    persistent.M  = A &&
    no attribute.M
}
```

To check if `UML` model `M` is consistent with a `RDBMS` model `N` the command **`check`** `{ uml2rdbms[M,N] }` is issued, with the scope of each signature being set to the number of elements of the respective class in each of the two models. Regarding enforce mode with GED minimization, in order to determine a new `UML` model `M'` consistent with `RDBMS` model `N`, with original model `M`, the command

```
run { uml2rdbms[M',N] && Delta_UML[M,M']=Δ }
```

is issued with increasing Δ values (starting at 0). In this case, the scope of each signature is set to the number of elements of the respective class plus Δ, to allow complete freedom in the choice of edit operations. The calculation and increment of both Δ and the scope is performed automatically by our tool. Since we are dealing with exact scopes, the class hierarchy must also be taken into consideration. For instance, for a `HSM` solution with one `CompositeState` and one `State`, the scope of `State` must be set to 2.

Regarding enforce mode with OBD minimization, the command

```
run { uml2rdbms[M',N] && M = first && M' = last }
```

is issued with increasing scopes Δ (plus one) for signature `UML`, as all `UML` atoms will belong to the total order entailed by the operations. Singleton fields `first` and `last` denote the first and last atoms of the `next` total order: they are constrained to be the original and updated model, respectively, meaning that the latter should be obtained from the former using Δ edit operations. The scope of the remaining signatures is inferred from the operations specified in the meta-model, allowing a finer control over the scopes of the model finder, since we know the behavior of all possible update steps. This requires the creation of elements by the operations to be detected, which is by itself an ambiguous issue in OCL-specified operations. For our technique, we assume that every new element created by an operation is identified with the **`oclIsNew()`** operation in the post-condition and inside a **`one`** quantification (a predicate which holds for exactly one element [42, p. 170]). With **`oclIsNew()`** tags inside other quantifiers we would not be able to precisely measure the scope increment. For instance, consider the operation `addAttribute(n:String)` from the `Class` class. Its post-condition would contain, among others, the following constraint:

```
self.attributes->one(a |
    a.oclIsNew() and a.name = n)
```

This in turn would be translated to Alloy as:

```
a not in self.(attributes.m) &&
self.(attributes.m') = self.(attributes.m) + a &&
a.(name.m') = n
```

The user is required to specify an upper-bound for Δ that limits the search for consistent targets. If several consistent models are found at the minimum distance our tool warns the user and allows him to see the different alternatives. If the user then desires to reduce such non-determinism, he can, for example, add extra OCL constraints to the meta-model or narrow the set of allowed edit operations to target a specific class of solutions. Section 6 will present a concrete example of how such narrowing can be done.

4 Bidirectionalizing ATL

ATL [24] is a widely used model transformation language created to answer the original QVT RFP, and thus shares some characteristics with the standardized QVT languages. Unlike QVT-R, ATL has *de facto* standard operational semantics implemented as a plugin for the Eclipse IDE[1]. However, it is unidirectional, in the sense that a transformation between M and N meta-models (which will be denoted by $\overrightarrow{t} : M \rightarrow N$, as it is a deterministic procedure), only specifies how to create an N model from an M model. The prescribed method to obtain bidirectional transformations with ATL is to write two unidirectional transformations. Unfortunately, this leads to obvious correctness and maintenance problems, since the language provides no means to check that they are inverses of each other, nor to automatically derive one from the other. Moreover, that only works well for essentially bijective transformations, since, unlike in QVT-R, in ATL transformations are not able to recover missing information from the previous target model, as they only receive the source model as input (as specified in the type of $\overrightarrow{t}$) In this section we explore how our technique can be adapted to confer bidirectional semantics to ATL transformations.

4.1 ATL language

ATL is a hybrid language with both declarative and imperative constructs. The authors advocate that transformations should be declarative whenever possible, and imperative specifications should only be used if specifying the transformation declaratively proves to be difficult [24]. In this work we restrict ourselves to declarative constructs, disregarding imperative ones (known in ATL as *called* rules and *do* blocks). Giving semantics to imperative constructs in Alloy's relational logic is doable (see, for example, [38]), but the need to explicitly represent all intermediate updates to the models in execution traces would deem our solver based approach completely unfeasible, in particular in presence of loops.

[1] http://www.eclipse.org/atl/

The main constituents of ATL transformations are *rules*, the ATL equivalent to QVT-R relations, whose abstract syntax is:

```
[[unique] lazy] rule R {
    from a : A (πM)
    to b : B (ϕ)
}
```

Rules consist of a single source element a from the source meta-model and a set of target elements (for the scope of this presentation we restrict ourselves to a single target element b) from the target meta-model. Source elements are selected by an OCL pattern π_M over their properties, while target patterns consist of bindings ϕ over the target element, which may consider values from source elements. Roughly, the execution semantics creates a target element for every source element that matches π_M. Source models are read-only and target models write-only, and thus transformations are not able to take into consideration existing elements in the target model, not even being able to "check-before-enforce" (although more recent work on incremental executions ATL executions could eventually be used to address this issue [26]).

Default rules are called *matched rules* and must be executed for all elements of the source model (similarly to QVT-R top relations). *Lazy rules*, unlike matched rules, are only executed if explicitly called from other rules (similarly to QVT-R non-top relations). They can either be *unique* or not: in unique lazy rules a source element is always matched to the same target element, no matter how many times the rule is called over that source element; in non-unique lazy rules a new target element is created every time it is called.[Nao suportamos non-unique lazy rules, certo? Nao devia ficar claro?]

One particular characteristic of ATL is that target bindings may rely on implicit traceability links between elements created by other matched rules. Target elements may be directly "assigned" source elements, in which case traces are used to retrieve the corresponding target element. This is possible because execution is divided in two phases: the first phase binds source elements to the source patterns and creates the target elements, implicitly creating traces between them; the second phase applies the bindings to the target elements, resorting to the traces if necessary. Since a source element cannot be matched by more than one rule [2], it is always possible to retrieve a single target element from

```
module hsm2nhsm;
create nhm : NHSM from hsm : HSM;

// StateMachineToStateMachine
rule M2M {
  from
    hm : HSM!StateMachine ()
  to
    nm : NHSM!StateMachine ( name <- hm.name )
}

// StateToState
rule S2S {
  from
    hs : HSM!State ( hs.container->isEmpty() )
  to
    ns : NHSM!State (
      name    <- hs.name,
      machine <- hs.machine
    )
}

// TransitionToTransition
rule T2T {
  from
    ht : HSM!Transition
  to
    nt : NHSM!Transition(
      source  <- ht.source->closure(container)->
        any(s | s.container->isEmpty()),
      target  <- ht.target->closure(container)->
        any(s | s.container->isEmpty()),
      machine <- ht.machine
    )
}
```

Fig. 7: The `hsm2nhsm` ATL transformation.

a source element. Lazy rules must be explicitly called and thus are not be taken into consideration in implicit resolutions.

Figure 7 presents an ATL version of the `hsm2nhsm` transformation using transitive closure. Rule `S2S` relates every top-level state in `HSM` to a state in `NHSM` with the same name, while rule `T2T` maps every transition in `HSM` to a transition on `NHSM` between the top-level containers of its source and target states. These are retrieved by filtering the result of the closure operation by a select operation.Note how in `T2T`, states of `HSM` the input model are being attributed to the source and target of transitions in the `NHSM` output model: the rule is taking advantage of the implicit traces created by `S2S` between `HSM` and `NHSM` states. A similar situation occurs in `S2S` when assigning state machines previously bound by `M2M`.

4.2 Overview of the Bidirectionalization Technique

To bidirectionalize ATL transformations we will first derive a consistency relation $\boldsymbol{T} \subseteq M \times N$ from an ATL transformation $\overrightarrow{\boldsymbol{t}} : M \to N$, and then use it to determine suitable (inverse) transformations according to the least-change semantics proposed in Section 2.4 for QVT-R enforce mode.

Since we are given the forward transformation $\overrightarrow{\boldsymbol{t}} : M \to N$, one could imagine that it would sufice to derive a suitable backward transformation $\overleftarrow{\boldsymbol{t}} : M \times N \to N$, thus lifting ATL transformations to the framework of *lenses* [12], as attempted before by Sasano et al [45]. A well-behaved lens consists precisely of a pair of transformations $\overrightarrow{\boldsymbol{t}} : M \to N$ (usually known as *get*) and $\overleftarrow{\boldsymbol{t}} : M \times N \to M$ (usually known as *put*), that is *acceptable*, $\overrightarrow{\boldsymbol{t}}\ (\overleftarrow{\boldsymbol{t}}\ (m, n)) = n$, guaranteeing that an update on n is indeed propagated to m, and *stable*, $\overleftarrow{\boldsymbol{t}}\ (m, \overrightarrow{\boldsymbol{t}}\ (m)) = m$, the lens equivalent to hippocraticness. The lens framework is designed to deal with transformations that are abstractions (i.e., surjective transformations), as implied by the asymmetric nature of the two transformations: the view n can always be derived solely from a source m as it contains less information. In particular, if a model n is updated to n' that falls outside the range of $\overrightarrow{\boldsymbol{t}}$, the behavior of $\overleftarrow{\boldsymbol{T}}$ is undefined. Such well behaved lens could be obtained in our least-change maintainer framework, by setting the forward transformation as an implicit consistency relation as $\boldsymbol{T}\ (m, n) \equiv n = \overrightarrow{\boldsymbol{t}}\ (m)$.

Unfortunately, this imposes some undesirable limitations in the allowed usage scenarios. Consider a very simple example where a source model `World` consists of a set of `Person` elements with a `name`, and a target model `Company` consists of a set of `Employee` elements with a `name` and (optional) `salary` (Fig. 8), and a trivial ATL transformation `employ` that maps every `Person` to an `Employee` with the same `name` and an empty `salary` (Fig. 9). This transformation is clearly not surjective since it only targets the subset of `Company` models where `Employee` elements have no `salary`. Now, consider a model wrd : `World` with a single person p and the corresponding model cpn : `Company` created by $\overrightarrow{\texttt{employ}}$. If the user updates cpn to cpn' by assigning a `salary` to p, there will be no valid wrd' such that $cpn' = \overrightarrow{\texttt{employ}}\ wrd'$, and thus cpn' would be an invalid update. This limitation would greatly reduce the updatability of the framework.

A possible solution to this problem would be to weaken the lens laws, as suggested in [45], by allowing $\overleftarrow{\boldsymbol{t}}\ (m, n)$ to produce a source m' whose view $n' = \overrightarrow{\boldsymbol{t}}\ (m')$ is not n (breaking acceptability) as long as propagating n' backward produces m' again, i.e.,

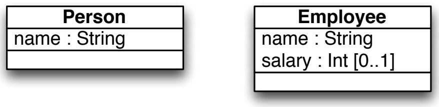


Fig. 8: Class diagrams of the `World` and `Company` metamodels.

```
module employ;
create cpn : Company from wrd : World;

// PersonToEmployee
rule P2E {
  from
    p : World!Person ()
  to
    e : Company!Employee ( name <- p.name )
}
```

Fig. 9: The `employ` ATL transformation.

$$\overleftarrow{t}\ (m,n) = m' \Rightarrow \overleftarrow{t}\ (m, \overrightarrow{t}\ (m')) = m'$$

(deemed *weakly acceptable*). However, even if the above update is now allowed, if the user updates the `World` model (for example, inserting a new person) and wishes to propagate such change to the `Company`, the forward transformation $\overrightarrow{\texttt{employ}}$ would erase the previously assigned salary of p, since it is not incremental. Embedding ATL in a lens framework with such weakened laws presumes that once $\overrightarrow{t}$ is run to generate a new target model from a source, subsequent updates can only be safely propagated backwards.

To overcome this limitation we opt instead to embed ATL transformations in the framework of maintainers, likewise to QVT-R. The main idea is to infer from $\overrightarrow{t} : M \to N$ a consistency relation $\boldsymbol{T} \subseteq M \times N$ such that every model m is considered consistent with any model that extends $\overrightarrow{t}\ (m)$, in the sense that it sets values for properties not bound by $\overrightarrow{t}$. This of course implies that $\overrightarrow{t} \subseteq \boldsymbol{T}$. From $\boldsymbol{T}$ a new forward transformation $\overrightarrow{\boldsymbol{T}} : M \times N \to N$ and a backward transformation $\overleftarrow{\boldsymbol{T}} : M \times N \to M$ can then be derived to propagate updates in both directions, using the least-change semantics described in Section 2.4 (obviously satisfying both the correctness and hippocraticness laws). Back to the example from Fig. 9, since $\overrightarrow{\texttt{employ}}$ does not bind the `salary`s in `Company` models, the `World` with a single `Person` p would be consistent with any `Company` with a single `Employee` p, whatever his `salary`. The following section will present a technique to infer one such possible $\boldsymbol{T}$ from $\overrightarrow{t}$.

The bidirectional ATL framework obtained with this technique satisfies the following properties. First, since $\overrightarrow{t} \subseteq \boldsymbol{T}$, the consistency relation trivially holds for pairs of models (m, n) such that $\overrightarrow{t}\ (m) = n$. Second, if $\overrightarrow{t}$ is surjective, applying either $\overrightarrow{t}$ or $\overrightarrow{\boldsymbol{T}}$ to an updated source will yield the same updated target: in this case, $\overrightarrow{t}$ completely defines the target elements, thus a model m is only related to $\overrightarrow{t}\ (m)$ by $\boldsymbol{T}$. In this case, the pair of transformations $\overrightarrow{t}$ and $\overleftarrow{\boldsymbol{T}}$ will form a well-behaved lens. In contrast, for non-surjective ATL transformations this is no longer the case. It is easy to see why by considering the toy example from Fig. 9: by updating *cpn* to *cpn′* with the insertion of a `salary`, *wrd* and *cpn′* will still be consistent by `Employ`, and thus neither $\overrightarrow{\texttt{Employ}}$ nor $\overleftarrow{\texttt{Employ}}$ will update the models; applying $\overrightarrow{\texttt{employ}}$ however would revert *cpn′* back to *cpn*. In this case $\overrightarrow{\texttt{employ}}$ and $\overleftarrow{\texttt{Employ}}$ do not form a well-behaved lens, satisfying only weak acceptability.

The derivation of a maintainer $\boldsymbol{T}$ does not invalidate the use of $\overrightarrow{t}$ paired with $\overleftarrow{\boldsymbol{T}}$: the pair comprises a stable and weakly acceptable lens. Due to non-incrementality of $\overrightarrow{t}$ however, $\overrightarrow{t}$ is better suited to initially create the target model from a source, at which point $\overrightarrow{\boldsymbol{T}}$ and $\overleftarrow{\boldsymbol{T}}$ can be used to propagate updates in both directions to maintain consistency.

4.3 Inferring a Consistency Relation

At first glance, the semantics of an ATL transformation shares some similarities with the checking semantics of QVT-R described in Section 2.3: pattern matching is used to filter candidate source elements, and it resembles the *forall-there-exists* quantification pattern to relate source and target elements. There are also some apparent differences: it is a directional semantics, in the sense that the above *forall-there-exists* quantification in principle should only be checked in the direction of the target (the direction of the transformation), and the existential quantifier should be unique, that is, for all candidate source elements, there must exist *exactly one* target element built with the target bindings. However there are some subtle differences: as the following example will show, the *forall-there-exists* semantics cannot be realized using quantifiers, and explicit traceability links must be used instead; check must be also performed in the opposite direction to avoid spurious target elements.

Consider again the simple transformation from Fig. 9, and the following semantics for the rule `P2E` with quantifiers, using a notation similar to the one introduced in Section 2.3 for QVT-R:

$$\texttt{P2E}_{\blacktriangleright}\ (\mathit{wrd} : \texttt{World}, \mathit{cpn} : \texttt{Company}) \equiv \forall\ p : \texttt{Person} \mid (\exists!\ e : \texttt{Employee} \mid p.\texttt{name} = e.\texttt{name})$$

Given a source model *wrd* with two `Person` elements with the same `name` a, this semantics would force a consistent target model *cpn* with exactly one `Employee` with `name` a. This is obviously not the intended ATL semantics, as two `Employee` elements with the same `name` are created by the transformation, one for each source `Person`. Obviously, relaxing the uniqueness constraint of the existential quantifier will not solve the problem, as an arbitrary number of `Employee` elements would be allowed. The one-to-one mapping between candidate source and generated target elements cannot be realized by quantifiers, but through an explicit traceability relation between them. Moreover, this directional check does not guarantee that the only `Employee` elements in the target model are the ones created by the transformation, and some check in the opposite direction must be performed to ensure that every `Employee` originates from a `Person`. Such semantics can be encoded through a higher-order quantification as follows:

$$\begin{array}{l}\mathtt{P2E}_{\blacktriangleright}\ (\mathit{wrd} : \mathtt{World}, \mathit{cpn} : \mathtt{Company}) \equiv \\ \quad \exists\ \mathtt{P2E}_{\lhd\rhd} \subseteq \mathtt{Person} \times \mathtt{Employee} \mid \\ \quad\quad \forall\ p : \mathtt{Person} \mid \exists\,!\ e : \mathtt{Employee} \mid \\ \quad\quad\quad \mathtt{P2E}_{\lhd\rhd}\ (p, e) \wedge p.\mathtt{name} = e.\mathtt{name}\ \ \wedge \\ \quad\quad \forall\ e : \mathtt{Employee} \mid \exists\,!\ p : \mathtt{Person} \mid \\ \quad\quad\quad \mathtt{P2E}_{\lhd\rhd}\ (p, e) \wedge p.\mathtt{name} = e.\mathtt{name}\end{array}$$

That is, the transformation ensures that there exists a traceability relation $\mathtt{P2E}_{\lhd\rhd}$ between every `Person` (since $\pi_{\mathtt{World}}$ is empty) to a unique `Employee` with the same `name`, and vice-versa. Note that this semantics considers target elements that fall outside the range of $\overrightarrow{\mathtt{employ}}$, namely those that have the `salary` defined.

In general, the semantics of matched rule R can be specified as follows:

$$\begin{array}{l}R_{\blacktriangleright}\ (m : M, n : N) \equiv \exists\ R_{\lhd\rhd} \subseteq A \times B \mid \\ \quad \forall\ a : A \mid \pi_M \Rightarrow (\exists\,!\ b : B \mid R_{\lhd\rhd}\ (a, b) \wedge \phi)\ \ \wedge \\ \quad \forall\ b : B \mid (\exists\,!\ a : A \mid R_{\lhd\rhd}\ (a, b) \wedge \pi_M \wedge \phi)\end{array}$$

This defines $R_{\lhd\rhd}$ as a one-to-one relation between every candidate source element and a single corresponding valid target element. The first expression states that every $a : A$ that matches the pattern π_M must be related to a single $b : B$ with the bindings ϕ; the second expression states that every $b : B$ must be related to a valid $a : A$. The binding ϕ in a rule assigns to the target element values possibly from the source element: unlike in QVT-R target patterns, all variables in ϕ must be previously assigned in the source pattern π_M. As such, they are interpreted likewise to **`where`** conditions in QVT-R.

An implicit call that might occur in the right-hand-side e of a binding is handled as follows. If e has a primitive type or is a target element then it is directly translated to Alloy. If e denotes a source element of type A, we retrieve the matching target element of type B from the respective traceability relation $R_{\lhd\rhd}$ (notice that it is always possible to uniquely determine $R_{\lhd\rhd}$, since source elements of a given type are restricted to be matched by a single rule [2]). Traceabilities can also be implicitly called over collections, as in the `T2T` transformation. Our tool also supports these implicit calls for collections that are sets, the above procedures being applied for every element e in the set.

Although the semantics of unique lazy rules also relies on explicit traceability links, there is a subtlety that prevents their encoding in a similar way to (top) matched rules, namely, it is quite difficult to infer in static time what elements a unique lazy rule must relate—recall that they only create unique target elements when called from another rule. As such, the semantics of these rules will be divided in two parts. Given an unique lazy rule R, the following predicate will only enforce the correctness of the respective traceability relation, with the existence and uniqueness checks being deferred to the rule call:

$$\begin{array}{l}R_{\blacktriangleright}\ (m : M, n : N) \equiv \exists\ R_{\lhd\rhd} \subseteq A \times B \mid \\ \quad \forall\ a : A, b : B \mid R_{\lhd\rhd}\ (a, b) \wedge \pi_M \Rightarrow \phi\end{array}$$

Moreover, when a unique lazy rule R is called over an expression e, we insert the following additional constraints to ensure that the trace between e and the matched element b exists and is unique:

$$\exists\,!\ b : B \mid R_{\lhd\rhd}\ (e, b) \wedge \forall\ a : A \mid \mathcal{T}_U\ (a, b) \Rightarrow a = e$$

Where $\mathcal{T}_U$ denotes the union of all unique lazy traces. The first part of the conjunction states that e is uniquely matched to a b by $R_{\lhd\rhd}$, and the second that b is not being matched to any other element by any other rule.

Finally, for an ATL transformation T, we assume that two models are consistent if the above semantics holds for all (matched and unique lazy) rules $\mathcal{R}_T$:

$$\boldsymbol{T}\ (m : M, n : N) \equiv \forall\ R : \mathcal{R}_T \mid R_{\blacktriangleright}\ (m, n)$$

This semantics can be encoded in Alloy in a similar manner to that of QVT-R, as described in Section 3.3. The higher-order existential quantification that asserts the existence of the traceability relation $R_{\lhd\rhd}$ can be encoded by skolemization, by explicitly declaring an Alloy relation that represents it. This ends up being similar to the actual encoding of QVT-R, where an auxiliary relation was also declared to encode $R_{\rhd}$, albeit for a different reason, namely to support recursion. Non-unique lazy rules are currently not supported by our technique.

5 Deployment

The technique for bidirectional model transformations presented in this article has been implemented as part of the Echo framework[2], a tool for managing intra- and inter-model consistency. In this section we first briefly present Echo features and architecture, and then describe with more detail two of its key components: the model visualizer and transformation optimizer.

5.1 The Echo Framework

The focus of this framework is to help users develop and keep their models consistent. It supports both intra-model (i.e. consistency between a model and its meta-model) and inter-model consistency (relating several models via bidirectional transformations, the focus of this article). In both cases, Echo can detect and repair inconsistencies.

Concerning intra-model consistency, given a meta-model M with OCL constraints, Echo can automatically check if a model m is consistent with M, that is $m : M$. This can be done for newly created models or every-time the user updates an existing model. If consistency of model m is broken, for example because some of the OCL constraints is violated, then Echo can automatically suggest minimally repaired models $m' : M$ using the model finding procedure described in this article, that is, it can find consistent models m' at minimum GED or OBD from m. Various alternatives for repaired models are presented to the user in increasing distance to the original model, among which the user is able to choose the preferred one. To help the user choose the preferred model, they can be depicted as graphs by resorting to the Alloy visualizer, as seen in Fig. 10. For better readability, an Alloy theme is automatically inferred from the meta-models (as described in Section 5.2). A user-defined theme can also be provided if desired. To help kickstart model development, Echo can also be used to generate a new minimal model $m : M$ (notice that often models cannot be empty due to meta-model constraints), or to generate scenarios for meta-model validation, that is, models parameterized by particular scopes and/or additional OCL constraints targeting specific configurations.

Concerning inter-model consistency, given a QVT-R or ATL transformation T, from which consistency relation $\boldsymbol{T} \subseteq M \times N$ is inferred, and models $m : M$ and $n : N$, Echo can automatically check if m and n are $\boldsymbol{T}$-consistent, that is $\boldsymbol{T}\ (m, n)$. In the case of QVT-R it follows the standard-compliant checking semantics presented in Section 2.3. For ATL, the semantics described in Section 4.3 is used. Given a transformation $\boldsymbol{T} \subseteq M \times N$ and models $m : M$ and $n : N$ such that $\neg\, \boldsymbol{T}\ (m, n)$, Echo can perform a minimal update to one of the models to recover consistency, for example produce $n' : N$ such that $\boldsymbol{T}\ (m, n')$. This repair follows the enforcement semantics satisfying the principle of least-change, as described in Section 2.4. Likewise to intra-model consistency recovery, the user is able to choose the desired repaired model among all minimal consistent models. Finally, given a transformation $\boldsymbol{T} \subseteq M \times N$ and a model $m : M$, Echo can produce a minimal model $n : M$ such that $\boldsymbol{T}\ (m, n)$ (likewise for the opposite direction). This is useful at early phases of model-driven software development, when the user has developed a first version of the source model, from which he wishes to derive a first version of the target. Afterwards, updates can be performed and consistency recovered incrementally to any of the models, by resorting to the same transformation.

While also available as a command-line application, Echo's main distribution platform is as a plugin for the Eclipse IDE, which automates the features just presented. Echo's environment consists of a set models, conforming to OCL-annotated meta-models, and a set of inter-model constraints specified by QVT-R and ATL transformations. Each model is thus restricted by the intra-model constraint entailed by the meta-model and any number of inter-model constraints simultaneously. The Echo plugin was designed to be used in an *on-line setting*, in the sense that the consistency tests are automatically applied as the user is editing the models and, thus, updates are expected to be incremental, leaving the original models as unmodified as possible. Every time the user updates a model, the system automatically checks its consistency in relation to the other artifacts. If a model is deemed inconsistent, the plugin displays an inconsistency error and proposes possible fixes. As there may be more than one consistent model at minimal distance, Echo presents all possible models in succession, allowing the user to choose the desired one, at which time the update is effectively applied to the model instance. If none of the minimal solutions is chosen, Echo presents models at increasingly higher distances from the original.

The plugin is built on top of the *Eclipse Modeling Framework* (EMF)[3], and resorts to the *Model Development Tools* (MDT) component to process OCL formulas and to the *Model-to-Model Transformation* (MMT) component to parse QVT-R and ATL specifications.EMF prescribes Ecore for the specification of meta-models,

[2] Download and more information available at `http://haslab.github.io/echo` and in the tool demo [33].

[3] `http://www.eclipse.org/modeling/`

while model instances are presented as XMI resources. To enhance the meta-models with additional constraints, we follow the technique proposed by MDT, of embedding the OCL constraints in meta-model annotations. Both Ecore meta-models and XMI instances are translated to Alloy following the techniques from Section 3, so that the transformation engine described in this article can be applied.

To promote inter-operability, EMF processes models defined in an abstract syntax, which are persisted as XMI resources. Thus, as a model-to-model transformation tool over EMF, Echo is only able to directly process models represented in XMI, much like the other MMT components (QVT-R and ATL). Echo's core engine can also be used directly as a library, in which case models are expected to be already parsed into the EMF's abstract syntax. Nonetheless, EMF has a wide support for domain-specific languages presented in a concrete syntax, which can be directly harnessed by Echo. The currently prescribed mechanism to convert models from concrete to abstract syntax is through Xtext[4], a language processing framework that provides parser generators as well as full integration with the Eclipse IDE through custom code editors. As an example, to process QVT-R transformations, Echo translates QVT-R specifications following the standard's concrete syntax to EMF's abstract syntax by relying on the MMT functionalities built over Xtext.

5.2 Visualizing Model Instances

As just described, the user is able to choose the desired repaired model from the range of all minimal consistent solutions. Performing such choice over the concrete XMI files would not be user-friendly (even with the standard Eclipse's XMI editor), so instead we resort to the Alloy graph visualizer, where perceiving models is as easy as grasping graphs. However, in order to be better understandable by the user these graphs must be presented in a shape that resembles its model structure. The Alloy visualizer allows the definition of custom themes, and our tool automatically determines one such theme using the information available from the Ecore meta-models. Alloy's magic theme functionality [44] also tries to infer a suitable theme from an Alloy specification through a set heuristics. However, while some of the visualization properties determined by our technique are similar to those inferred by the magic theme, the extra information available in the meta-model, and knowledge about the underlying encoding of the transformations, proves to be an advantage and eliminates the need of said heuristics.

The most evident feature of the inferred theme is hiding the extra Alloy fields required by the underlying enforcing mechanism but irrelevant for the user, in particular the signature's state fields and the auxiliary fields used to represent relation calls. Our enforcing mechanism also requires that both the original source and target models, as well as the updated target model coexist in a single Alloy instance. Presenting them together to the user would be very confusing, so we opted to project the instance over concrete models, focusing first on the updated model, but allowing the user to visualize the others if he so desires (Alloy's magic theme would try to infer such projections [44], but our experiments showed that it would fail to pick the desired one in this particular case). To better highlight the differences between the original and the repaired models, the elements inserted or removed by the repair are painted in a different color (green), while the existing elements are painted gray. Calculating the GED Δ between Alloy instances already requires calculating the symmetric difference between their elements (and links). Since the Alloy visualizer allows subset signatures to be drawn differently, that component of the Δ function is reused to that end. Elements belonging to different classes are distinguished by shape.

Like in [44], enumeration literals are hidden and fields whose target type is an enumeration are presented as node labels rather than as edges to the enumeration literals. However, we need not use heuristics to detect enumerations, as their existence can be detected directly in the Ecore meta-model. Following the same reasoning, Alloy fields that originated from attributes in the meta-model are also presented as node labels rather than as edges to the attribute's value node, to minimize the number of visual elements. As a consequence of the projected model states, sets end up also being represented by node labels.

Finally, we are also able to determine a suitable spanning tree for the graph, that defines its dominant hierarchical structure. In our context, these are represented by the containment associations of the meta-model, which define the overall structure of the model instances, the remaining associations depicting only references between existing elements. In the Alloy visualizer spines are defined by tagging such fields as *influencing the layout*.

Figure 10 shows two models: the left-one conforms to the RDBMS meta-model and is depicted with the standard Eclipse XMI editor; the right-one conforms to the UML meta-model and is depicted with the embedded Alloy visualizer, using the automatically inferred theme.

[4] http://www.eclipse.org/xtext/

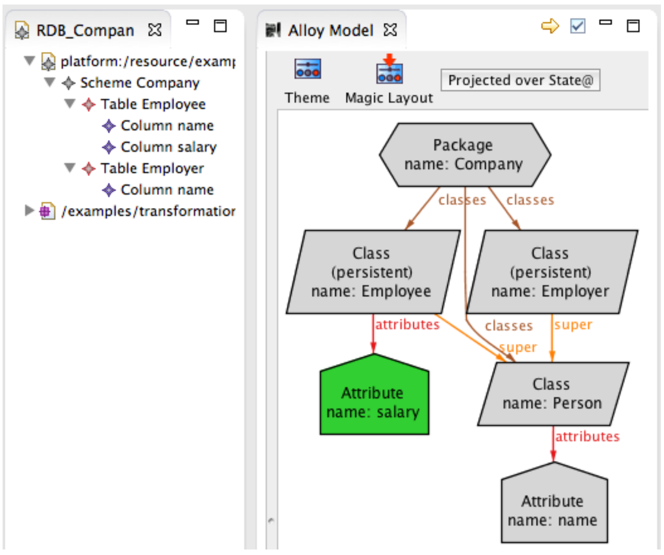


Fig. 10: A snapshot of Echo, with RDBMS and UML models depicted with Eclipse's XMI editor and with the embedded Alloy visualizer, respectively.

Note how the graph is adapted to the UML meta-model: different classes are shaped differently, while the attribute `name` and the set `persistent` are presented as labels rather than edges. All information not relevant to the presentation of the model is hidden. The class diagram is a very simple company model, where there are `Employee` and `Employer` classes, which are both extensions of `Person`. Each `Person` has a `name`, which is inherited by the persistent classes `Employee` and `Employer`.

The models in Fig. 10 are consistent with the QVT-R `uml2rdbms` bidirectional transformation, hence the relational scheme on the left-hand-side with the two corresponding tables. The `Employee` table has a `salary` column, whose matching attribute in the UML model is painted green (in contrast to the other elements painted gray). This means that this attribute has just been inserted by Echo in order to restore consistency between the two models.

Figure 11 presents models kept consistent by the `hsm2nhsm` transformation. The original HSM model was a simple state machine with two top-level states, `Idle` and `Active`. `Active` is a composite state, containing two sub-states, `Waiting` and `Running`, with a transition from `Idle` directly to `Waiting`. In the collapsed view, `Waiting` and `Running` are dropped, but the transition between the sub-state `Waiting` and the top-level state `Idle` is inherited by `Active`. It is worth noting that in this example HSM and NHSM are two different meta-models, and thus the different shape assigned to elements of similarly named classes. At some point, the

$$
\begin{array}{lll}
\mathsf{R\&S} \equiv \mathsf{R}, & \text{if } \mathtt{R}\ \mathbf{in}\ \mathtt{S} & (\cap\text{-Subset}) \\
\mathsf{R\&S} \equiv \mathsf{S}, & \text{if } \mathtt{S}\ \mathbf{in}\ \mathtt{R} & (\cap\text{-Subset}) \\
\mathsf{R{:}{>}A} \equiv \mathsf{R}, & \text{if } \mathbf{univ}.\mathtt{R}\ \mathbf{in}\ \mathtt{A} & (\rho\text{-Subset}) \\
\mathsf{A{<}{:}R} \equiv \mathsf{R}, & \text{if } \mathtt{R}.\mathbf{univ}\ \mathbf{in}\ \mathtt{A} & (\delta\text{-Subset})
\end{array}
$$

Fig. 12: Redundancy elimination.

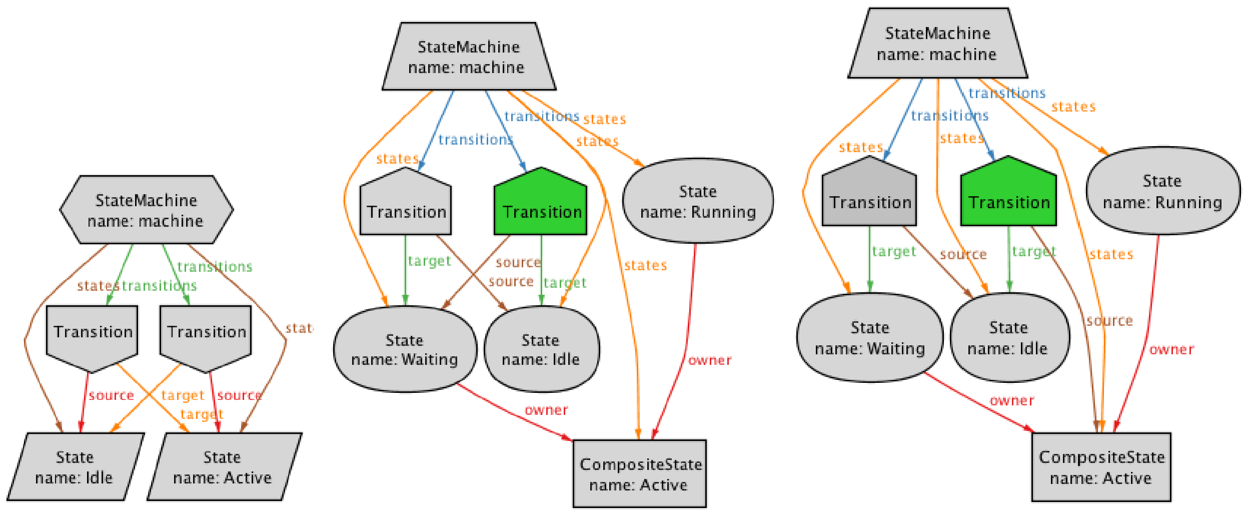
NHSM model was updated with the insertion of a transition from `Active` to `Idle`, breaking the consistency between the models. When propagating the update, Echo proposes three minimal solutions. Figure 11 presents two of them: set the composite state `Active` as the source of the new transition or choose instead one of its sub-states, in this case, `Waiting`. In the third minimal repair (not shown) the sub-state `Running` is set as the source of the new transition.

5.3 Optimizing Alloy Models

The major caveat of model finding approaches is scalability. While we are aware that our technique will never be as efficient as syntactic approaches (even if more expressive), in this section we present some optimizations that enable its application to many realistic examples. Although a novel contribution, the reader uninterested in Alloy technical details may skip this section as it does not affect the semantics of the proposed technique.

As presented in Section 2.3, QVT-R semantics relies heavily on nested *forall-there-exists* quantifications. These introduce inefficiency, since the complexity of the generated formulas may prevent skolemization and other optimizations performed by Kodkod [50] (the underlying relational model finder that supports Alloy) when translating to SAT. As such, the main goal of our optimization procedure is to eliminate (or reduce the scope of) as many quantifiers as possible, sometimes taking advantage of meta-model knowledge not readily available to Kodkod.

Figure 13 presents the equivalence laws used by our system (as rewriting rules oriented from left to right) to eliminate or reduce the scope of quantifiers. Among the most effective, we have the one point rules, that require as side-condition that the set over which the quantified variable ranges is a singleton. Using knowledge about the meta-model, this condition is many times trivial to check, namely when such set is the result of a navigation expression over a mandatory attribute. Figure 12 presents some additional laws that are used to eliminate redundant expressions, again using knowledge about the meta-model.



(a) `NHSM` model. (b) Possible `HSM` model. (c) Possible `HSM` model.

Fig. 11: `hsm2nhsm`-consistent models as presented in Echo.

To simplify the application of such rules, navigation expressions are kept normalized in the shape `x.R`, where `x` is typically a quantified variable and `R` an arbitrary composition of binary relations or their converse. Such normalization can be done by application of associativity and converse laws concerning the relational composition operator, such as `R.x` ≡ `x.~R` or `(~(R.S))` ≡ `(~S).(~R)`. Moreover, in this normalization we attempt to isolate the nearest quantified variable in a membership check using the rule `y` **`in`** `x.R` ≡ `x` **`in`** `y.(~R)` to potentiate the application of trading rules. Finally, whenever possible, we also replace multiplicity checks by their navigational equivalent, for example using the law **`some`** `x.R` ≡ `x` **`in`** `x.R.(~R)`.

As an optimization example, consider the most simple QVT-R transformation from `uml2rdbms`, namely relation `P2S` in the direction of `RDBMS`. By applying QVT-R semantics the following formula would result from our embedding in Alloy:

```
all p:package.m,n:String | n in p.(name.m) =>
  some s:schema.m' | n in s.(name.m')
```

Although simple, this formula already contains 3 quantifications whose range is loosely restricted (for instance, `n` is freely quantified over all strings). Figure 14 shows how this formula can be simplified using the above rules. To understand how the side-conditions can be easily checked using meta-model knowledge, consider the `name` attribute in the `Package` class of the `UML` meta-model. As we have seen in Section 3.2, when embedding this meta-model in Alloy this attribute is encoded as a relation of type `Package -> `**`String`**` -> UML` (to be used always as a binary relation in the context of particular `UML` model—in our optimization example the model `m`), constrained by the following multiplicity and inclusion dependency fact.

```
all m:UML | name.m in package.m -> one String
```

From this we can deduce that `p.(name.m)` **`in String`**, the side-condition required for the first application of rule ∩-Subset, that **`one`** `p.(name.m)`, the side-condition to the application of ∀-One-Point, and that the domain of `name.m` is a subset of `package.m`, formally `(name.m).`**`univ in`** `package.m`, in the final application of δ-Subset.

[This example may not be credible and is bugging me a little, because if the one point rule was applied before the expansion of **`some`** then the last universal quantifier could not be eliminated. Which leads to the obvious question about confluence...]

This optimization procedure essentially attempts to translate relational logic formulas to the, so-called, point-free notation: a version of this logic with no variables (nor quantifiers). Such notation is well-know for its amenability to proof and optimization through simple equational reasoning [39], and transformation of Alloy formulas to such style has been explored before [13,30], as means to perform unbounded verification proofs. In this case, its application to optimization is particularly effective, since it takes advantage of the fact that formulas originating from our embedding follow a very specific pattern, and information about the meta-model is readily available to speed-up side-condition checks.

Finally, some other optimizations, not related to quantifier elimination, are also performed. For example, when embedding meta-models in Alloy, fields are not created for associations marked as opposite of another

```
          (all x:none | R) ≡ true                                        (∀-Empty)
         (some x:none | R) ≡ false                                       (∃-Empty)
          (all x:A | true) ≡ true                                          (∀-Top)
       (all x : A | false) ≡ no A                                       (∀-Bottom)
       (some x : A | true) ≡ some A                                        (∃-Top)
        (some x:A | false) ≡ false                                      (∃-Bottom)
             (all x:A | R) ≡ R[x := A],   if one A                   (∀-One-Point)
            (some x:A | R) ≡ R[x := A],   if one A                   (∃-One-Point)
  (all x:A | x in B => R) ≡ (all x:A&B | R)                           (∀-Trading)
 (some x:A | x in B && R) ≡ (some x:A&B | R)                          (∃-Trading)
       (all x:A | x in B) ≡ A in B                                   (⊆-Def-Set)
   (all x:A | x.R in x.S) ≡ A<:R in S                                (⊆-Def-Rel)
```

Fig. 13: Quantifier elimination and restriction.

```
all p:package.m, n:String | n in p.(name.m) => some s:schema.m' | n in s.(name.m')  (∀-Trading)
all p:package.m, n:String&(p.(name.m)) | some s:schema.m' | n in s.(name.m')          (∩-Subset)
all p:package.m, n:p.(name.m) | some s:schema.m' | n in s.(name.m')               (Normalization)
all p:package.m, n:p.(name.m) | some s:schema.m' | s in n.~(name.m')                (∃-Trading)
all p:package.m, n:p.(name.m) | some s:(schema.m')&(n.~(name.m')) | true                (∃-Top)
all p:package.m, n:p.(name.m) | some (schema.m')&(n.~(name.m'))                     (∩-Subset)
all p:package.m, n:p.(name.m) | some n.~(name.m')                                (Normalization)
all p:package.m, n:p.(name.m) | n in n.~(name.m').(name.m')                      (∀-One-Point)
all p:package.m | p.(name.m) in p.(name.m).~(name.m').(name.m')                    (⊆-Def-Rel)
(package.m)<:(name.m) in (name.m).~(name.m').(name.m')                              (δ-Subset)
(name.m) in (name.m).~(name.m').(name.m')
```

Fig. 14: Optimization example.

existing association. Instead, when a call to an opposite association occurs in a formula (e.g. `namespace.m`), it is just replaced to a call on its opposite using the converse operator (e.g. ~(`classes.m`)). This further reduces the overall amount of variables and constraints during SAT solving.

Note that our tool performs these optimizations only once, when the meta-models and transformations are loaded and embedded into Alloy, and not every time the transformation is run after an update. As such, the time spent on the optimizations (which is almost negligible anyway) does not affect the performance of the least change update propagation procedure.

6 Evaluation

As made evident on Section 3.3, choosing between GED and OBD imposes a clear tradeoff between control over the updates and user overload due to the mandatory definition of operations. In this section we start by analyzing the consequences of this choice with concrete examples. We then discuss the impact of bidirectional (as opposed to unidirectional) checking semantics in QVT-R. Finally we present some efficiency tests to access the scalability of our technique.

6.1 GED vs. OBD

Consider the UML class diagram and database scheme from Fig. 10, and imagine that the database manager decides that employers also have salaries, creating a column `salary` in the `Employer` table. Since GED is meta-model independent, our tool automatically infers how to calculate model distance according to it. In particular, the minimal repairs on the `UML` model according to GED are either setting `Employee` as a super-class of `Employer` or to move the attribute `salary` from `Employee` up to `Person`, both at distance 2. If none of these are desirable repairs, the user can ask for the next closest solution at distance 3, which in this case is the introduction of a new attribute `salary` in `Employer`.

Suppose the user wants to rule out all repairs that change the class hierarchy or assign the same cost to either create a new attribute or move an attribute from one class to another. To do so, he can specify (using OCL) which are the valid edit operations that can be performed to repair a model. For our running example, we only assume the existence of operations whose signature is presented in Fig. 1. Notice that there are no edit operations that modify the hierarchy, and both creation and moving of an attribute are now atomic edit operations. Through OBD our technique finds the minimal sequence of edit operations that repairs the model. In our company running example, there will now be two minimal repairs, namely insert the new attribute `salary` in `Employer`, or moving the existing one from `Employee` to the common super-class `Person`, which were considered at different distances using GED. As expected, the solution which set `Employer` as a subclass of `Employee` has also been excluded.

For another example of tradeoff between GED and OBD, consider the expansion/collapse of state machines example. Imagine the user wants to allow the occurrence of errors and creates a new simple state `Error` on the collapsed diagram and a transition to it from `Active`. Propagating this update back to the expanded state machine using GED would yield 3 possible solutions at minimal distance: the creation of the simple state `Error` with a transition to it from either the composite state `Active` or the substates `Waiting` or `Running`, all at minimal cost. The user could easily navigate through these solutions and select the most suitable one depending on the context. [Fig. 11 could have depicted this example.][Realmente...]

Suppose however that the user prefers that transitions inserted in the collapsed state machine are applied only at the top-level states. If that is the case, besides the `addTopState` and operations `addSubState`, he could simply define an `addTransition` operation in a way that it only allows the insertion of transitions between top-level states. Figure 3 presents the signatures of these edit operations. In that case, if the previous update was propagated using the OBD metric, it would present only one solution at minimal distance, namely the insertion of a new transition from the composition state `Active` to `Error`.

We believe this combination of a meta-model independent metric and a user parameterizable one provides a high level of flexibility to our technique.

6.2 Bidirectional vs. Unidirectional Checking Semantics

While the prevalent idea is that the QVT-R standard forces checking semantics to be bidirectional (i.e., run the test in both directions) [4], this requirement may be too strong in some contexts. In fact, some ambiguities in the standard allow different interpretations and ModelMorf [49], the tool that allegedly follows the QVT-R standard the closest, allows unidirectional checks. In this section we briefly analyze the consequences of this directionality.

Let us consider the `uml2rdbms` transformations, and compare a unidirectional consistency check in the direction of `UML` and a bidirectional check as prescribed by the QVT-R standard. The main difference between them is that, while in the bidirectional version, only extra classes not matched by any relation are disregarded (insertion of a non-persistent class does not introduce inconsistencies), in the unidirectional version any extra class not related to a table is disregarded (insertion of a class, even if persistent, never causes inconsistencies). Clearly, this is undesirable in `uml2rdbms` transformation, and would be as well in `hsm2nhsm`.

However, this is not always the case. Consider for instance the consistency relation between UML class diagrams and UML sequence charts. One of the basic consistency constraints between these models is that all classes mentioned in the sequence chart must exist in the class diagram; however, not all classes in the class diagram must exist in the sequence chart. This kind of consistency relation would be impossible to specify with QVT-R's *forall-there-exists* bidirectional checks, unless the classes which are mentioned in the sequence chart were somehow (artificially) marked in the class diagram, so that a pattern to filter them out can be defined in the consistency relations.

Our tool's default checking mode uses a bidirectional semantics, but the fact that it consists of the conjunction of the two unidirectional tests makes it easy to adapt the system to perform unidirectional tests when desirable.

In fact, this asymmetry issue is just one dimension of a more general problem that emerges when considering multi-directional transformations. In [32] we show that our embedding can be trivially generalized to the multi-directional scenario, where updates on multiple models are propagated to a set of designated target models (another feature not currently offered by any existing QVT-R tool). We also show that the QVT-R standard enforcement semantics for multiple models—the *forall-there-exists* constraint from every source model to a

n	nodes	edges	variables
2	18	27	449
3	25	39	763
4	32	52	1063
5	39	63	1469
6	46	75	1941
7	53	87	2479
8	60	99	3083
9	67	111	3753
10	74	123	4489

Table 2: Scalability tests size for enforce mode with GED and $d = 1$.

single target model—is too restrictive, excluding many interesting application scenarios.

6.3 Scalability

At the time of writing, no benchmark for the assessment of bidirectional transformation tools has been proposed. Thus, to assess the scalability of our technique we devised a class of synthetic examples of the familiar `uml2rdbms` transformation, with the intention of achieving linear increases both in model size (number of nodes and edges when seen as a graph) and required update distance.

The shape of a `UML` class diagram of dimension n consists of a spine of n non-persistent `Class` elements (identified as class i at level i), each with a persistent sub-`Class` (identified as class i' at level i), which have themselves a single `Attribute` (with the same name i' as the owning class). Thus, a `UML` model of dimension n has $5n + 2$ nodes and $8n$ edges. For instance, Fig. 15 depicts the `UML` model for $n = 3$, the number of nodes being the number of model elements (10) and string literals (7), while the number of edges is the number of association links (14) and element's attributes (13), the latter shown as node labels in the Fig. 15. The corresponding `RDBMS` models, to be `uml2rdbms`-consistent, must contain a `Table` with a single `Column` for each persistent `Class` i', that is, $3n+2$ nodes and $4n+1$ edges. Since the `UML` and `RDBMS` models coexist, the total size of the environment is the sum of the respective sizes, with the exception of string literals which are shared. These models were generated using `Echo`'s model generation feature, which allows the specification of model sizes and the definition of extra OCL constraints that parametrize the shape of the generated solutions.

To introduce inconsistencies, new `Column` elements are inserted in the `RDBMS` model that impose repairs on the `UML` model. The smallest inconsistency consists of inserting in `Table` n' a `Column` $(n-1)'$. To solve this inconsistency, the minimal update is to move `Attribute`

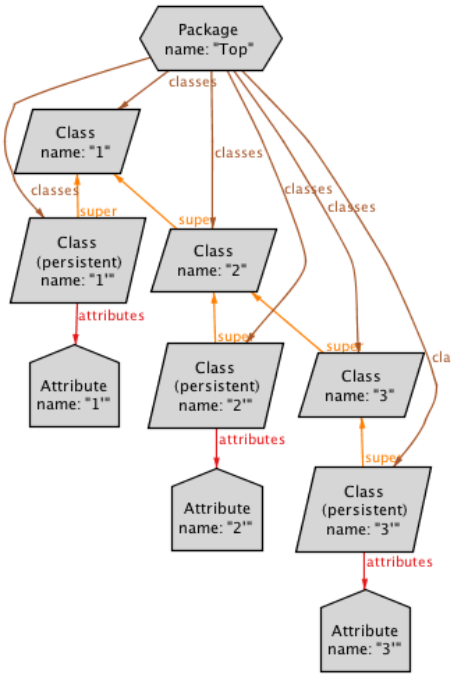


Fig. 15: Synthetic `UML` model with $n = 3$.

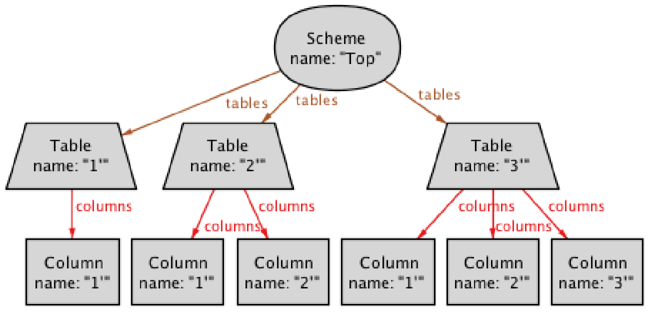


Fig. 16: Synthetic `RDBMS` model with $n = 3$ and $d = 2$.

$(n-1)'$ in the `UML` model to the $(n-1)$ non-persistent `Class`, so that it is shared by both `Class` n' and `Class` $(n-1)'$. This has a cost $\Delta = 2$ for GED and $\Delta = 1$ for OBD. Increasingly distant updates $d < n$ can be attained by inserting in every `Table` i' such that $i > n-d$ every `Column` j' such that $n-d \leqslant j < i$, resulting in updates $\Delta = 2d$ for GED and $\Delta = d$ for OBD in the `UML` model. Thus, an inconsistency at distance d introduces $\frac{d(d+1)}{2}$ nodes and $d(d+1)$ edges. As an example, Fig. 16 presents the `RDBMS` model for $n = 3$ with inconsistencies for $d = 2$.

All tests were run using `Echo` over `Alloy` 4.2 with the MiniSat solver, on a 1,8 GHz Intel Core i5 with 4 GB memory running OS X 10.8. We performed experiments for models up to $n = 10$ and update distance up to $d = 3$, when applicable. Table 2 summarizes the total size of both models for n up to 10 given an update $d = 1$. The last column represents the number of variables present in the SAT problem generated by Kodkod, Alloy's underlying relational model finder [50], when repairing the consistency between both models. All tests were run multiple times as to get the average performance values.

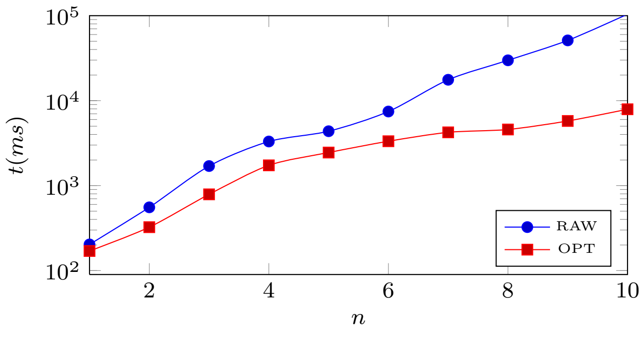
Figure 17 compares execution times (shown in log scale) of runs with and without the optimizations presented in Section 5.3. Figure 17 compares checkonly runs, and the gains are very significant. For $n = 10$ the optimized version takes only 7% of the time spent by the non-optimized version, with average gains of 45%. Checkonly runs do not require the measurement of model distances, so the choice of the distance metric does not affect the performance. Figure 17b compares enforcement runs using GED and OBD again with and without formula simplifications, for a fixed $d = 1$. The optimized versions are in average 29% and 56% more efficient than the non-optimized versions, for GED and OBD respectively, but again the difference grows fast, and for $n = 10$ the optimized versions take only 7% of the execution time of the non-optimized ones for both GED and OBD. As already mentioned in Section 5.3, optimizing Alloy formulas may take some time, but since this optimization is performed only once at static-time (when the transformations are translated to Alloy), it does not affect the time effectively spent in the repair. The gain from enforcement executions using GED to those using OBD is also significant (in average the first takes 75% of the time of the second, and around 40% for $n = 10$), but these results should be analyzed with caution, as they occur in a controlled scenario where GED repairs require only twice as much solving iterations than the ones with OBD. In practice, OBD can be much faster than GED if each atomic operation is more complex, combining multiple insertions/deletions of nodes and edges, allowing inconsistencies to be repaired with smaller distances.

Figure 18 depicts the execution times of checkonly and enforcement modes (with optimizations), using both GED and OBD, respectively, as the dimension n of the model increases, and for different fixed update distances d. Execution times for $d = 0$, i.e., consistency checks, take up to $8s$ for $n = 10$. While these values are not competitive against other existing techniques for consistency checking, they are due to the lack of support for instances of Alloy: partial solutions must be encoded by additional singleton signatures and constraints in the model. The performance in this case could be significantly improved by embedding the technique directly in Kodkod or by using Alloy extensions with support for partial instances, like the one proposed in [37], as our current studies show [10]. The impact on running times of the increasing d is intrinsic to our iterative technique, since every Δ step requires a new model finding run. This is better depicted in Fig. 19 that presents the same data but in relation to increasing distance Δ, for fixed model dimensions n.

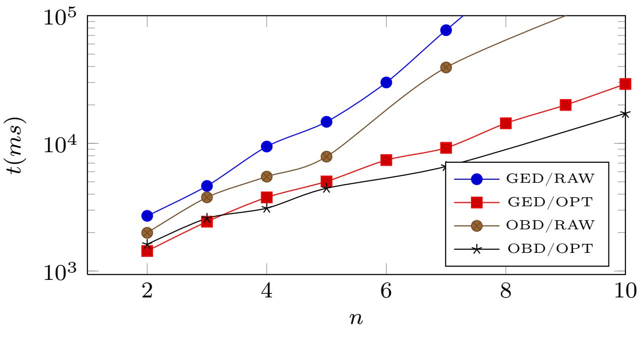
Although not ready to handle industrial-size models, Echo's greatest strength lies in its ability to allow the user to quickly and simply analyze and debug transformation specifications. In fact, a great challenge in model transformation is to guarantee that the behavior of the specified artifacts reflects the intention of the user: with a predictable least-change semantics and quick provision of feedback to the user, Echo excels in these tasks. In this context, the size of the models is not as crucial—as put by the small scope hypothesis advocated by the Alloy creators [23], most problems on specifications may be flagged by small instances. The fact that we were able to detect heretofore undetected problems in the standard's `uml2rdbms` transformation attests this. Nonetheless, despite the size of the models, the complexity introduced by the meta-models and inter-model constraints could alone deem solving unfeasible—in fact, without the optimizations presented in Section 5.3 that was precisely the case. In the future we intend to develop functionalities dedicated to automatically check specific properties of model transformations—like the fact that they are total, deterministic, or that they always produce well-formed models—to fully exploit this facet of Echo. As our technique is already based on model finding, these extensions are rather straight-forward to implement.

7 Related Work

QVT-R Tool Support Regarding tool support for QVT-R transformations, Medini and ModelMorf are the main existing functional tools. Medini [22] is an Eclipse plugin for a subset of the QVT-R language. Although popular, its (unknown) semantics admittedly disregards the semantics from the QVT standard (it does not have a checkonly mode, for instance). To support incremental executions, it stores explicit traces between elements of the two models. ModelMorf [49] allegedly follows the QVT standard closely (although once again the concrete semantics is unknown), since its development team was involved in the specification of the standard. However, the development of the tool seems to have stopped. None of these tools has support for OCL constraints on the meta-models. Other prototype tools have been proposed but once again the implemented semantics are not completely clear. Moment-QVT [3] is an Eclipse plugin for the execution of QVT-R transformations by resorting to the Maude rewriting system; [28] proposes

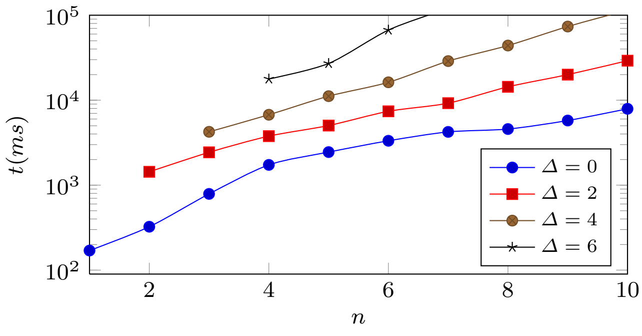


(a) Checkonly.

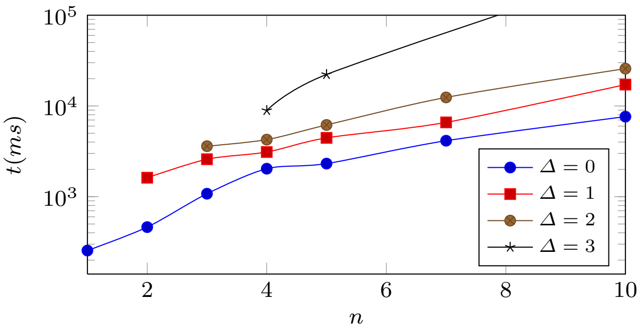


(b) Enforcement ($d = 1$) for GED and OBD.

Fig. 17: Execution times for optimized (OPT) and non-optimized (RAW) implementations.

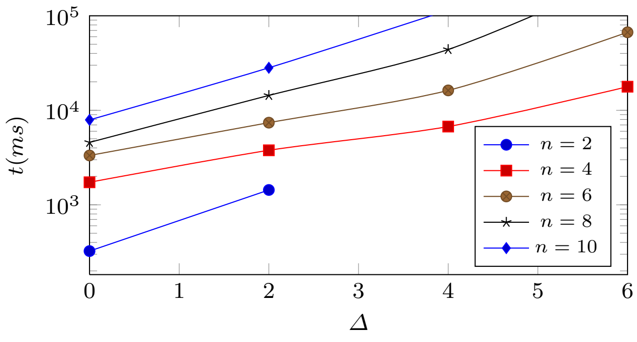


(a) GED.

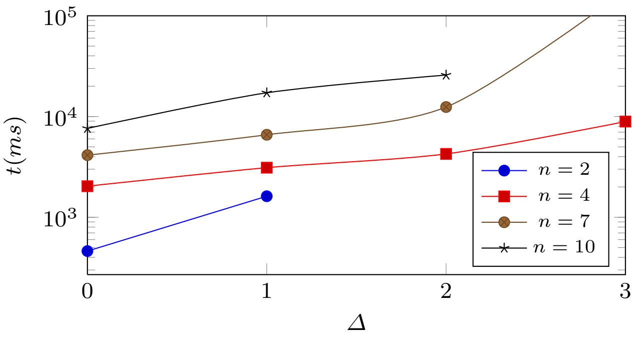


(b) OBD.

Fig. 18: Execution times over model size n, for fixed Δ values.



(a) GED.



(b) OBD.

Fig. 19: Execution times over model distance Δ, for fixed n values.

the embedding of QVT-R in Colored Petri Nets. All these tools support only unidirectional transformations, in the sense that they ignore the original target model. As such, they are not able to retrieve information not present in the source, leading to the generation of completely new models every time the transformation is applied. Once again, none supports OCL constraints on the meta-model. In [16] the authors discuss the possible implementation of QVT-R transformations in TGGs. While some TGGs tools prevent loss of information by supporting incremental executions [15] or partial matches between domains [17,20], this embedding focuses only on the embedding of QVT-R specifications in TGG frameworks, disregarding the consequences on the enforcement semantics.

A technique that follows an approach similar to ours is the JTL tool [8], although it does not support QVT-R, but rather a restricted QVT-like language. Like ours, JTL generates models by resorting to a solver (the DLV solver), which is able to retrieve some (unquantified) information from the original target. However, it is not clear how the solver chooses which information to retrieve or how the new model is generated. It also forces the totality of the transformation, returning inconsistent models in case there is no consistent one.

QVT-R Semantics Recently there has been an attempt to formalize the standard QVT-R enforce semantics [5], following previous work on the checking semantics [47, 4]. As prescribed in the standard, to enforce the *forall-there-exists* semantics, the procedure consists of a creation phase of new target elements whenever a source element does not have a matching target (or modification of existing ones, if keys are used), followed by a deletion phase, to remove target elements that are no longer matching a source element. These phases occur only at top-level relations, as **when** and **where** are assumed to be predicates that top-level quantified elements must comply. This procedure does not take into consideration additional constraints on the meta-model, in particular no specific technique is proposed to fill-in mandatory attributes and associations of newly created (or modified) elements, taking into account such meta-model constraints, and **when** and **where** clauses (likewise to our technique, the usage of solvers is hinted as a possible solution). Since it closely follows the standard, this semantics also suffers from the problems already described in Section 2.4.

Two approaches have been proposed for the validation of QVT-R transformations that also rely on solvers. In [14] the authors use Alloy to verify the correctness of QVT-R specifications, in order to guarantee the well-formedness of the output and avoid run-time errors. In [7] OCL invariants of the shape *forall-there-exists* are inferred from QVT-R transformations (much like the checking semantics), that allow the validation of QVT-R specifications under a set of properties. It supports OCL constraints in the meta-model and recursive calls are translated to recursive OCL specifications. A similar technique has also been developed for the verification of ATL transformations [6]. However, these approaches are not focused on enforce mode and its semantics, and do not analyze the behavior of the transformation for concrete input models, which is the focus of our embedding. As already stated, our technique could also be adapted to support the validation of similar properties, like checking if a transformation is injective or that all consistent models are well-formed.

Bidirectionalization of ATL The relation between ATL and QVT has previously been explored in [25]. However, the focus was on aligning the architecture of the two languages, without any semantic considerations. They suggest that interoperability could be attained by mapping both QVT-R and ATL to QVT Operational.

Some previous work has been done towards the bidirectionalization of ATL. In [52] the authors infer a synchronization procedure from a subset of the byte-code produced by the ATL compiler, in the sense that both the source and target model can be updated in order to restore consistency. However, it is not clear how the restrictions on the byte-code are reflected on the ATL language. In [45] the authors interpret models as graphs and ATL declarative rules as UnCAL operations over said graphs, which are bidirectionalized in the GRoundTram bidirectional graph transformation system [21]. However, the supported subset of the ATL language is much more limited than ours, namely matching rules cannot have source patterns and target bindings must comply to a very limited OCL subset, excluding rule calls (implicit or explicit). As discussed in Section 4.2, if we see our ATL bidirectional transformations as lenses, the bidirectional properties from [45] also hold in our framework. None of these approaches are concerned with enforcing least-change updates.

Solver-based Model Repair Some research have been made on applying model finding techniques to *model repair* problems, in particular using Alloy. Model repair and bidirectional model transformations are closely related, since the two meta-models can be merged in a single global meta-model, where inter-model consistency could be specified by standard intra-model constraints, with the tweak that, when propagating an update to the target, model repairs should only be allowed in the subset of the global merged model conforming to the original source meta-model. Even if existing model repair tools do not implement this tweak, and thus cannot be directly applied to bidirectional model transformations, the underlying repair techniques are quite similar and both areas could benefit from crossbreeding.

In [27] the authors propose a general approach for constraint-based solving in the context of MDE (including application to model repair), using the Alloy and OPL solvers as concrete examples. However, the original inconsistent model is specified as the lower bound for the new model, meaning that the solver will only be able to add new atoms and relations while solving the constraints. Following a similar approach, [48] assesses the feasibility of using Kodkod to repair inconsistencies. Given an inconsistent Kodkod problem, a consistent problem is found by relaxing the bounds of the original model in order to allow the addition of new relations or the removal of relations suspected of causing the selected inconsistencies. This assumes that the concrete inconsistencies were previously detected by an external tool. In both approaches there is also no control over how close the new model is to the original one and the authors do not reason on how to manage the creation of new atoms. [19] describes a technique for generating quick fixes for DSMLs embedding on CSP over models. The technique guarantees that the number of inconsistencies on the model decreases, even if side-effects occur. This is achieved by applying every candidate fix to the inconsistent model and detecting and counting the inconsistencies in the resulting model. In terms of expressivity, this last approach is the closest to ours, but, being also solver based, it suffers from the same scalability issues.

Least-change Transformations As far as we are aware, studies on least-change bidirectional transformations are limited to the seminal work of Meertens [35], as already discussed in Section 2.4, and our own previous work on composing least-change lenses [34]. While it would be interesting to explore the compositionality of QVT-R transformations under the least-change principle, classic maintainers are already known not to be compositional [35], greatly lowering the expectations for a composable least-change QVT-R language.

Our least-change approach, like QVT-R, is state-based, in the sense that transformations (and thus the model distance metrics) consider only the pre- and post-states of the model, in contrast to operation-based frameworks [11], where transformations are provided extra information about how the updated model was attained, either through the applied edit sequence or through a sameness relation between model elements. This extra knowledge allows the transformations to disambiguate overlapping scenarios, the classic example being the ability to distinguish modifications from deletions/insertions, which under state-based frameworks are undistinguishable unless elements are assigned unique keys.

8 Conclusions and Future Work

This article proposed a QVT-R bidirectional model transformation tool, supporting both the standard checking semantics and a clear and precise enforcement semantics based on the principle of least change. It also supports meta-models annotated with OCL constraints and specification of allowed edit operations, which allows its applicability to non-trivial domains and provides a fine-grained control over non-determinism. The implementation is based on an embedding in Alloy, taking advantage of its model finding abilities. We have also extended our tool to support the bidirectionalization of ATL transformations. The tool is deployed as a plugin for the Eclipse IDE, focusing on user-friendliness, namely presenting in a clear way which model repairs are being applied.

Being solver-based, the main drawback of the proposed tool is performance. Improving it is the main goal of our future work: we intend to explore incremental solving techniques to speed-up the execution of successive commands with increasing scope, and to define mechanisms to infer which parts of target model can be fixed *a priori* in order to speed-up solving. In particular, we are currently analyzing the impact of embedding our technique directly in Kodkod, which has support for partial instances, and adapting it to rely on Max-SAT solvers instead, through the use of target-oriented solving techniques [10]. Nonetheless, even in its present status the tool is already fully functional, much due to the development of optimization techniques. In particular, it already proved effective in debugging existing transformations, namely helping us unveiling several errors in the well-known object-relational mapping that illustrates QVT-R specification. In the future we plan to further explore the debugging aspect of the tool by providing means to automatically verify and validate correctness properties of model transformations.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling **9**, 69–86 (2010)
2. ATLAS group: ATL user guide. `http://wiki.eclipse.org/ATL/User_Guide`
3. Boronat, A., Carsí, J., Ramos, I.: Algebraic specification of a model transformation engine. In: FASE'06, *LNCS*, vol. 3922, pp. 262–277. Springer (2006)
4. Bradfield, J., Stevens, P.: Recursive checkonly QVT-R transformations with general when and where clauses via the modal mu calculus. In: FASE'12, *LNCS*, vol. 7212, pp. 194–208. Springer (2012)
5. Bradfield, J., Stevens, P.: Enforcing QVT-R with mu-calculus and games. In: FASE'13, *LNCS*, vol. 7793, pp. 282–296. Springer (2013)
6. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: ICFEM'12, *LNCS*, vol. 7635, pp. 198–213. Springer (2012)
7. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software **83**(2), 283–302 (2010)
8. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: JTL: a bidirectional and change propagating transformation language. In: SLE'10, *LNCS*, vol. 6563, pp. 183–202. Springer (2010)
9. Cunha, A., Garis, A., Riesco, D.: Translating between Alloy specifications and UML class diagrams annotated with OCL. Software and Systems Modeling pp. 1–21 (2013)
10. Cunha, A., Macedo, N., Guimarães, T.: Target oriented relational model finding. In: FASE'14, *LNCS*, vol. 8411, pp. 17–31. Springer (2014)

11. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: The symmetric case. In: MoDELS'11, *LNCS*, vol. 6981, pp. 304–318. Springer (2011)
12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007)
13. Frias, M.F., Pombo, C.L., Aguirre, N.: An equational calculus for Alloy. In: ICFEM'04, *LNCS*, vol. 3308, pp. 162–175. Springer (2004)
14. Garcia, M.: Formalization of QVT-Relations: OCL-based static semantics and Alloy-based validation. In: MDSD Today 2008, pp. 21–30. Shaker Verlag (2008)
15. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and System Modeling **8**(1), 21–43 (2009)
16. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. Software and System Modeling **9**(1), 21–46 (2010)
17. Greenyer, J., Pook, S., Rieke, J.: Preventing information loss in incremental model synchronization by reusing elements. In: ECMFA'11, *LNCS*, vol. 6698, pp. 144–159. Springer (2011)
18. Guerra, E., de Lara, J.: An algebraic semantics for QVT-relations check-only transformations. Fundam. Inform. **114**(1), 73–101 (2012)
19. Hegedüs, Á., Horváth, Á., Ráth, I., Branco, M.C., Varró, D.: Quick fix generation for DSMLs. In: VL/HCC'11, pp. 17–24. IEEE (2011)
20. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Software and System Modeling pp. 1–29 (2013)
21. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In: ASE'11, pp. 480–483. IEEE (2011)
22. ikv++ technologies ag: Medini QVT. `http://projects.ikv.de/qvt/`
23. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, revised edn. MIT Press (2012)
24. Jouault, F., Kurtev, I.: Transforming models with ATL. In: MoDELS'05 Satellite Events, *LNCS*, vol. 3844, pp. 128–138. Springer (2005)
25. Jouault, F., Kurtev, I.: On the architectural alignment of ATL and QVT. In: SAC'06, pp. 1188–1195. ACM (2006)
26. Jouault, F., Tisi, M.: Towards incremental execution of ATL transformations. In: ICMT'10, *LNCS*, vol. 6142, pp. 123–137. Springer (2010)
27. Kleiner, M., Fabro, M.D.D., Albert, P.: Model search: Formalizing and automating constraint solving in MDE platforms. In: ECMFA'10, *LNCS*, vol. 6138, pp. 173–188. Springer (2010)
28. de Lara, J., Guerra, E.: Formal support for QVT-Relations with Coloured Petri Nets. In: MoDELS'09, *LNCS*, vol. 5795, pp. 256–270. Springer (2009)
29. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes **31**(3), 1–38 (2006)
30. Macedo, N.: Translating Alloy specifications to the pointfree style. Master's thesis, Escola de Engenharia, Universidade do Minho, Braga, Portugal (2010)
31. Macedo, N., Cunha, A.: Implementing QVT-R bidirectional model transformations using Alloy. In: FASE'13, *LNCS*, vol. 7793, pp. 297 – 311. Springer (2013)
32. Macedo, N., Cunha, A., Pacheco, H.: Towards a framework for multidirectional model transformations. In: EDBT/ICDT'14 Workshops, *CEUR Workshop Proceedings*, vol. 1133, pp. 71–74. CEUR-WS.org (2014)
33. Macedo, N., Guimarães, T., Cunha, A.: Model repair and transformation with Echo. In: ASE'13, pp. 694–697. IEEE (2013)
34. Macedo, N., Pacheco, H., Cunha, A., Oliveira, J.N.: Composing least-change lenses. ECEASST **57** (2013)
35. Meertens, L.: Designing constraint maintainers for user interaction. In: Third Workshop on Programmable Structured Documents. Tokyo University (2005)
36. Milicevic, A., Jackson, D.: Preventing arithmetic overflows in Alloy. In: ABZ'12, *LNCS*, vol. 7316, pp. 108–121. Springer (2012)
37. Montaghami, V., Rayside, D.: Extending Alloy with partial instances. In: ABZ'12, *LNCS*, vol. 7316, pp. 122–135. Springer (2012)
38. Near, J.P., Jackson, D.: An imperative extension to Alloy. In: ASM'10, *LNCS*, vol. 5977, pp. 118–131. Springer (2010)
39. Oliveira, J.N.: Extended static checking by calculation using the pointfree transform. In: LerNet'08, *LNCS*, vol. 5520, pp. 195–251. Springer (2009)
40. OMG: MOF 2.0 Query/View/Transformation specification (QVT), version 1.1 (2011). `http://www.omg.org/spec/QVT/1.1/`
41. OMG: OMG Unified Modeling Language (UML), version 2.4.1 (2011). Available at `http://www.omg.org/spec/UML/2.4.1/`
42. OMG: OMG Object Constraint Language (OCL), version 2.3.1 (2012). Available at `http://www.omg.org/spec/OCL/2.3.1/`
43. OMG: OMG Meta Object Facility (MOF), version 2.4.1 (2013). Available at `http://www.omg.org/spec/MOF/2.4.1/`
44. Rayside, D., Chang, F.S.H., Dennis, G., Seater, R., Jackson, D.: Automatic visualization of relational logic models. ECEASST **7** (2007)
45. Sasano, I., Hu, Z., Hidaka, S., Inaba, K., Kato, H., Nakano, K.: Toward bidirectionalization of ATL with GRoundTram. In: ICMT'11, *LNCS*, vol. 6707, pp. 138–151. Springer (2011)
46. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and System Modeling **9**(1), 7–20 (2010)
47. Stevens, P.: A simple game-theoretic approach to check-only QVT relations. Software and System Modeling **12**(1), 175–199 (2013)
48. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod model finder for resolving model inconsistencies. In: ECMFA'11, *LNCS*, vol. 6698, pp. 69–84. Springer (2011)
49. Tata Research Development and Design Centre: ModelMorf. `http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`
50. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS'07, *LNCS*, vol. 4424, pp. 632–647. Springer (2007)
51. Voigt, K.: Structural graph-based metamodel matching. Ph.D. thesis, University of Desden (2011)
52. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE'07, pp. 164–173. ACM (2007)