Analysis of Source-to-Target Model Transformations in QueST

Hamid Gholizadeh, Zinovy Diskin, Sahar Kokaly, Tom Maibaum
Computing and Software Department, McMaster University, Canada
{mohammh | diskinz | kokalys | maibaum}@mcmaster.ca

Abstract

Query Structured Model Transformation (QueST) is a framework for defining source-to-target Model Transformations (MTs) in a structured and declarative manner. In this paper, we study how MT-related properties can be analyzed in QueST. Each MT in QueST is equivalent to a logical theory and checking properties for the MT is equivalent to analyzing correctness of such properties (i.e., demonstrating that they are theorems) in this theory. We will explain the idea by encoding an underlying theory of a simple MT example as an Alloy specification and defining three sample properties. We show that the correctness of one of the properties is refuted by the Alloy analyzer, and provide a manual proof for the other two.

1 Introduction

Model Transformations are growing large in size and complexity in real world MDE applications. Source-to-target Model Transformations (henceforth referred to as MTs) constitute an important class of transformations and employing formal mathematical techniques for their analysis is desirable in order to gain confidence in their expected properties. QueST (Query Structured Model Transformation) is a formal specification framework for defining MTs in a setwise (rather than elementwise) declarative manner based on queries. An MT definition in QueST is similar to how relational views are defined in the SQL world, i.e., by giving query results new names. By defining queries, we unveil the hidden information in the source model that is needed for the generation of the target model elements, and thus can map the latter to queries against the source model. The basics of QueST are explained in our previous work [GDM14, Dis09], while in the present paper we aim to show how QueST can be used for the analysis of *MT-related* properties[1].

The basic idea of our approach to property checking is straightforward: an MT definition $\boldsymbol{D}$ is encoded as a logical theory $\boldsymbol{Th}(\boldsymbol{D})$ in an appropriate logic, a property to be checked is encoded as a logical proposition (sentence) $\boldsymbol{P}$, and checking the property for $\boldsymbol{D}$ amounts to checking the validity of semantic entailment $\boldsymbol{Th}(\boldsymbol{D}) \models$

[1]Here we follow the terminology developed in [ACL$^+$14], which distinguishes language-related property checking, such as termination and determinism, vs. MT-related property checking that is specific to each MT definition.

$\boldsymbol{P}$, which means that all execution instances of $\boldsymbol{D}$ satisfy the property. For such a semantic validity check, we can use a model checker or an instance generator like Alloy (for a limited scope analysis). If the logic used for encoding is complete, we can replace semantic validity by syntactic provability, $\boldsymbol{Th}(\boldsymbol{D}) \vdash \boldsymbol{P}$, so the problem would be equivalent to proving or disproving the correctness of proposition $\boldsymbol{P}$ in the theory $\boldsymbol{Th}(\boldsymbol{D})$. This process can be performed manually, or by receiving some assistance from a theorem prover.

The very idea of logical encoding of MTs for property checking is not new [BECG12, CLST11]. However, two features of QueST make it very well suited for the realization of this idea. First, an MT definition in QueST is itself close to its logical encoding as a theory: we can see a query language as an algebraic version of the corresponding logic. For example, the well-known equivalence of relational algebra and first-order logic means that any relational query can be reformulated as a FOL formula. Second, QueST's MT definitions are amenable to being built in a structured and modular way [GDM14], and their structural properties are directly transformed to their logical encoding.

To explain our technique, we will use a simple MT example and three related properties, and encode them in Alloy, which provides both a convenient language for specifying logical theories, and facilities to support their analyses. Note, however, that QueST's MT analysis method is independent of the implementation tool like Alloy.

The paper is organized as follows. In Section 2, we introduce our running MT example and three properties, and explain how MTs are defined in QueST. In Section 3, we present a logical encoding of the example in the Alloy language, and then analyse the properties introduced in Section 2: disprove one of them using the Alloy analyzer, and manually prove the other two. Related work is discussed in Section 4, and Section 5 concludes.

2 Running Example

In this section, we present a simple MT example and some properties that we are interested in checking about this transformation. We first describe the transformation in natural language, and then specify it in QueST.

2.1 HappyPeople: A Model Translation Example

The transformation should translate models specifying people who own and like cars to models of happy people who drive vehicles. The source and the target metamodels are depicted with light gray squares in Fig. 1 (the target metamodel on the right and the source metamodel on the left). Boxes and arrows in the metamodels denote classes and associations, and red bracketed expressions indicate constraints (ignore the green dashed elements for the moment). There are two requirements for the translation. The first is that cars and vehicles are considered to be synonyms: every car in a source instance should be a vehicle in the target, and conversely. The second is a (very modest) criterion of happiness: a person in the source is considered to be happy if there is at least one car that she both likes and owns, and then she is allowed to drive such a car (or cars). We will use this MT as our running example, and refer to it as the HappyPeople MT. Suppose we want to check the following three input-output properties:

- ***Property 1***: Every happy person drives at least one vehicle.
- ***Property 2***: All vehicles are driven by somebody.
- ***Property 3***: Happy persons only drive vehicles/cars they like.

Note that while Properties 1 and 2 are formulated entirely in the language of the target metamodel, Property 3 involves both metamodels and assumes backward traceability of happy persons and vehicles to persons and cars, respectively.

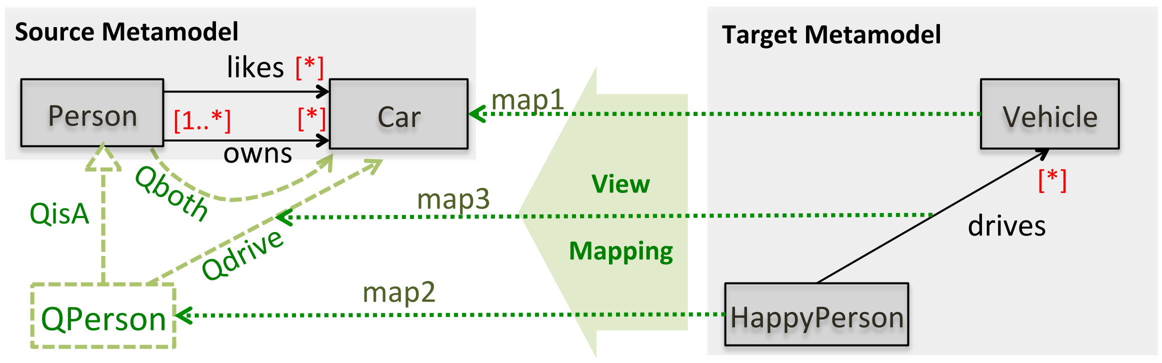


Figure 1: QueST definition of the HappyPeople MT

2.2 Specifying HappyPeople MT in QueST

The entire Fig. 1 demonstrates the HappyPeople MT definition in QueST. An MT definition in QueST is guided by the target metamodel: each element (a class or an association) in the target metamodel is to be mapped to a corresponding element (class or association, respectively) in the source metamodel. For example, class *Vehicle* is mapped to class *Car* (see Fig. 1) because they should be isomorphic, given the requirements of the MT. However, the source metamodel does not have elements directly corresponding to the class *HappyPerson* and the association *drives*, and the QueST idea is to reproduce such "missing" elements by finding suitable queries against the source metamodel, which result in exactly the elements needed to complete the mapping. Thus, for the HappyPeople MT, we need to find queries against the source metamodel, which would derive a class and an association corresponding to the class *HappyPerson* and the association *drives* in the target metamodel. We build such queries in two steps described below as queries Q1 and Q2.

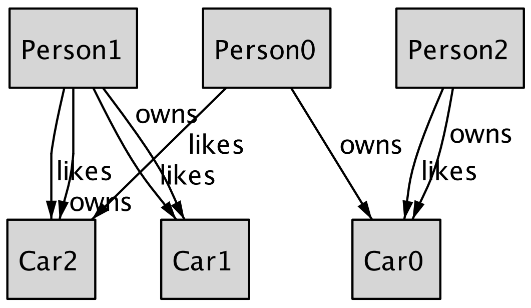
To ease understanding of our query descriptions, we switch to the terminology of relational theory, in which classes and associations are referred to by their intended semantic interpretation as sets and (binary) relations, respectively.

1. ***Q1: Relation Intersection.*** This query takes relations *owns* and *likes*, and produces their intersection relation *Qboth* between the same source and target sets (note the dashed curved arrow in Fig. 1). Thus, this query augments the metamodel with a new arrow *Qboth*.

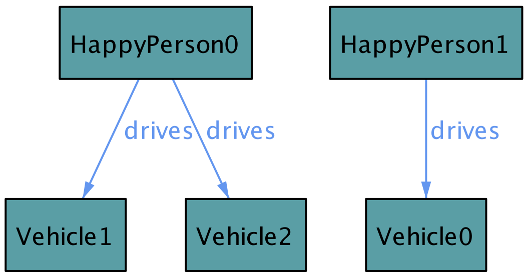
2. ***Q2: Domain Restriction.*** The query takes relation *Qboth* as its input, and produces (i) a subset of *Person* called *QPerson* for which the relation *Qboth* is defined (note the inclusion/subclassing arrow *QisA* from *Person* to *QPerson* denoting the subseting relation), and (ii) the restriction of *Qboth* to *QPerson* called *Qdrives* (in fact, *Qdrives* $=$ *Qboth* as a set of pairs, but their domains are different).

As mentioned in Sec. 1, queries in QueST have a well-defined logical semantics. In Section 3.1, we will specify the precise semantics of the above queries in Alloy.

After the source metamodel is extended with required derived elements, we can complete the MT definition mapping as shown in Fig. 1 (we also call it the *view mapping*, as what such a mapping defines is exactly a view whose schema is given by the target metamodel). The mapping is shown as a block arrow comprising three individual links that we call maps: *map1*, *map2*, *map3*. Syntactically, the mapping is a graph morphism from the target metamodel graph to the augmented source metamodel graph. Semantically, the mapping comprises a set of bijections between the elements in the target and the source metamodels. At the time of MT execution, the three map arrows specify how the elements of a specific type are generated in the target. For example,



(a) Input Source model



(b) Generated target model

Figure 2: Instance generation

all the elements of *HappyPerson* in the target are generated from the elements of *QPerson* produced by the corresponding query.

Fig. 2(a) is a source model, and Fig. 2(b) is its corresponding target model generated by the above QueST MT definition[1]. Indeed, as each of *Person1* and *Person2* owns and likes the same car (i.e., *Person1* owns and likes *Car1* and *Car2*, and *Person2* owns and likes *Car0*), they appear as happy people in the target model. The cars they both own and like also appear in the target as vehicles they drive. A general specification of the QueST execution mechanism can be found in [Dis09, GDM14].

3 Property Analysis

Query definition language underlying QueST is a parameter: for implementation, it is to be substituted by some concrete language, say SQL, or OCL. In addition, we need a language to encode the view definition mapping. Both, the mapping and queries (operations) can be encoded with constraints. For example, to encode a binary operation, we introduce a ternary predicate R with a constraint that for any x and y, there is one and only one z such that $R(x, y, z)$ holds true. Encoding of mappings will be explained below. Thus, as soon as we have a sufficiently expressive logical language L (e.g., FOL, OCL, or another appropriate alternative), an MT definition in QueST can be encoded as a theory in L. In this section, we show how the HappyPeople MT can be presented as an Alloy specification; i.e., we use Alloy as L. Then we present encoding of properties in Alloy, and show how we can check them.

3.1 QueST MT definition as an Alloy Theory

3.1.1 Background.

Alloy enables quick development of specifications and provides facilities for their analysis [Jac12]. Alloy's primary concepts are sets and (binary) relations, so that any system is modelled as a collection of sets and relations with a number of constraints defined over them. Relations are understood both *extensionally* (as sets of ordered pairs, $R \subseteq A \times B$) and *navigationally* (as partial multi-valued mappings, $R : A \nrightarrow B$; below we will use stroked arrows for relations and ordinary arrows $f : A \to B$ for functions, i.e., single-valued relations). The major operations of the extensional view are ordinary set union, intersection and complement. The major operation of the navigational view is relation composition: relations $R_1 : A \nrightarrow B$ and $R_2 : B \nrightarrow C$ can be composed into $R_1.R_2 : A \nrightarrow C$ with the following semantics: $\forall a : A, \forall c : C, (a, c) \in R_1.R_2 \iff \exists b : B, (a, b) \in R_1 \wedge (b, c) \in R_2$. Alloy provides these relational algebraic operations as well as the operation of transitive closure. Moreover,

[1]These instances are generated by the Alloy tool after encoding of the above QueST MT definition in Alloy. We will discuss the encoding process in Sec. 3.1.

Alloy also provides notation for writing FOL formulas with logical connectives and quantifiers. Thus, the same constraint or query can be specified in several different ways, providing great flexibility in using the language.

In the following, we explain how we encode different components of a typical QueST MT including metamodels, queries and mappings, and combine them in a unified Alloy specification. Because of the Alloy specific syntax for specifying relations, it is not possible to completely separate the encoding of these components in the syntax; however, by adding comments inside the specification we will provide a distinctive view representing each component encoding.

```
open util/relation
//------ Source metamodel -------------
sig Person {likes,owns:set Car,
        Qboth:set Car}          //query definition
sig Car{}
fact{ all c:Car | some p:Person | c in p.owns}

//------ Target metamodel and view-mapping ---------
sig HappyPerson {drives:set Vehicle,
        map1: QPerson}  //the mapping
sig Vehicle{
        map2: Car}      //the mapping

//the view-mapping constraints
fact{
        //map1
        bijection[map1,HappyPerson,QPerson]
        //map2
        bijection[map2,Vehicle,Car]
        //commutivity constraints ensuring map3 (see Lemma 1)
        drives.map2=map1.Qdrives
}

//------ Query definitions  -------------
//Relation Intersection Query
fact {Qboth = likes & owns}

//Domain Restriction Query
sig QPerson in Person{ Qdrives :set Car }
fact {QPerson =  Qboth.univ }
fact { Qdrives = Qboth }
```

Figure 3: Alloy Specification of HappyPeople MT definition in Fig. 1

3.1.2 Encoding Metamodels.

Alloy's view of the world as consisting of sets and relations perfectly matches the understanding of metamodels in QueST. Therefore, the source and the target metamodels of an MT can be directly encoded in Alloy. In Fig. 3, lines 3,5 and 6 encode the source metamodel (see Fig. 1), and lines 9 and 11 encode the target metamodel. Classes become signatures, and associations become relations defined inside curly brackets of their corresponding domain signature; e.g., $likes : Person \nrightarrow Car$ and $owns : Person \nrightarrow Vehicle$ are two relations defined under signature *Person*. The keyword *set* in these relations' definitions specifies the multiplicity constraints 0..*. Line 6, specifies the multiplicity constraint 1..* at the tail of the owns relation in the source metamodel (see Fig. 1).

3.1.3 Encoding Queries.

We encode each query operation by adding to the Alloy specification a family of sets and relations produced by the query. The *Relation Intersection* Query produces the relation *Qboth* (see line 4), and the *Domain Restriction* Query produces the set *QPerson* and the relation *Qdrives* (see line 29). The *QisA* relation in Fig. 1 is encoded implicitly by the *in* keyword at line 29, denoting the subsetting relation between *QPerson* and *Person*. The semantics of the query operations are reflected as constraints over the query results. For the Intersection Query, line 26 ensures straightforward intersection semantics for *Qboth*. Lines 30-31 specify the corresponding constraints for the Domain Restriction Query results. $QPerson = Qboth.univ$ at line 30 makes *QPerson* equal to projection of *Qboth* onto *Person*, and $Qdrives = Qboth$ at line 31 is in fact a concise encoding of $Qdrives = QisA \cdot Qboth$, since *QisA* is an identity relation.

3.1.4 Encoding the mappings.

In the HappyPeople example, the *view mapping* consists of three mapping links or just *maps* connecting the target metamodel with the source metamodel. Semantically, if a map links two classes, (like *map1* and *map2*), then it specifies a bijection between the corresponding sets, and we add to our Alloy specification the corresponding facts (see lines 16-19). If a map links relations like $map3$, semantically it means a bijection between the relations as sets of pairs, and moreover, the following *commutativity* constraint: if $map3(h, v) = (p, c)$, then $map1(h) = p$ and $map2(v) = c$. However, Alloy does not allow us to define a relation between relations like *map3*, and we need to find a work-around.

Lemma 1. *Let $R_i : A_i \nrightarrow B_i$, $i = 1, 2$ be two relations, and $f : A_1 \to A_2$, $g : B_1 \to B_2$ two functions such that the following* commutativity *condition holds: $R_1.g = f.R_2$. Then there is a uniquely defined function $f * g : R_1 \to R_2$ between relations as sets such that for a pair $(a, b) \in R_1$, $f * g(a, b) = (f(a), g(b))$. Moreover, if functions f, g are bijections, function $f * g$ is a bijection as well (and in this case we say that relations R_1 and R_2 are isomorphic).*

Proof. Straightforward. □

Thus, the commutativity condition above implies the existence of the required mapping between associations, and adding it to the Alloy spec (see line 21 in Fig. 3) completes the encoding of the view-mapping. In this way, the entire Alloy specification in Fig. 3 encodes our sample QueST MT definition in Fig. 1. In the next section, we show how MT properties are translated into the Alloy language and checked.

3.2 Analyzing Properties

Having the specification in Fig. 1 for the HappyPeople example, we can formally define the properties listed in Sec. 2.1. Fig. 4 lists the precise definition of these properties in Alloy; they are defined inside assertion blocks that let us use the Alloy analyzer to check their validity. The Alloy analyzer works based on scope parameters;

that is, assertions are checked within a user-defined scope that limits the maximum number of elements of each set. In the case that Alloy finds a counterexample within the defined scope, it means the assertion is not valid; otherwise, the assertion is valid within the scope, but *might* be valid beyond it.

We checked the three properties with the number 20 assigned to the scope parameters of the sets. The analyzer did find a counterexample for the second assertion refuting its correctness: there might be some vehicles without a driver in the generated models. Fig. 5 demonstrates one of these counterexamples generated by the Alloy analyzer. In the figure, the source model (gray elements), the generated model (blue elements), query results (green elements), and the traceability links (red arrows labeled with *map1* and *map2*) are all presented explicitly. *Vehicle1* in the generated model that comes from *Car1* is not driven by anybody.

```
//--------Property definition-----------
//Prop1 : everybody who is happy drives at least one vehicle.
assert ass1{
        all p:HappyPerson | some v:Vehicle | p->v in drives
}
//Prop2: all vehicles are driven by somebody
assert ass2{
        all v:Vehicle | some h:HappyPerson | h->v in drives
}
//Prop3: if somebody drives a vehicle he likes that vehicle
assert ass3{
        all h:HappyPerson | all  v:Vehicle |
                                h->v in drives => map1[h]->map2[v] in likes
}
```

Figure 4: Properties specified in Alloy

For the first and the third property, Alloy could not find any counterexamples, meaning that they *might* be valid even for an unlimited scope. One way to increase certainty about their validity is to increase the scope parameter value to a higher number; but no matter how much we increase the scope boundaries, the uncertainty will never disappear. To gain greater confidence, we need to provide a formal proof (which gives us absolute confidence modulo the adequacy of the formal encoding of the subject matter). This could be done manually, or semi-automatically with a theorem prover, but in general provability of FOL statements is undecidable; the latter of course does not prevent the existence of decision procedures for specific fragments and cases.

In the following, we will manually prove both of the above two properties to demonstrate the approach.

Lemma 2. *Two isomorphic relations (see Lemma 1) have the same head and tail multiplicities.*

Proof. Straightforward. □

Now let $\boldsymbol{Th}$ denote the relational theory encoding of HappyPeople MT as specified in Fig. 3.

Theorem 1 (Property 1 Holds)**.** *Every happy person drives at least one vehicle, formally:* $\boldsymbol{Th} \models (\forall h{:}\mathit{HappyPerson}, \exists v{:}\mathit{Vehicle}) v \in h.\mathit{drives}$.

Proof. In fact, we need to prove that the multiplicity of relation *drives* is [1..*]. However, relations *drives* and *Qdrives* are isomorphic by the construction of $\boldsymbol{Th}$ (lines 16-21 in Fig. 3), and so by Lemma 2 we need to prove the [1..*]-multiplicity of relation *Qdrives*. As the relation *Qdrives* is total by construction (line 30 in Fig. 3), relation *Qdrives* does have multiplicity [1..*]. □

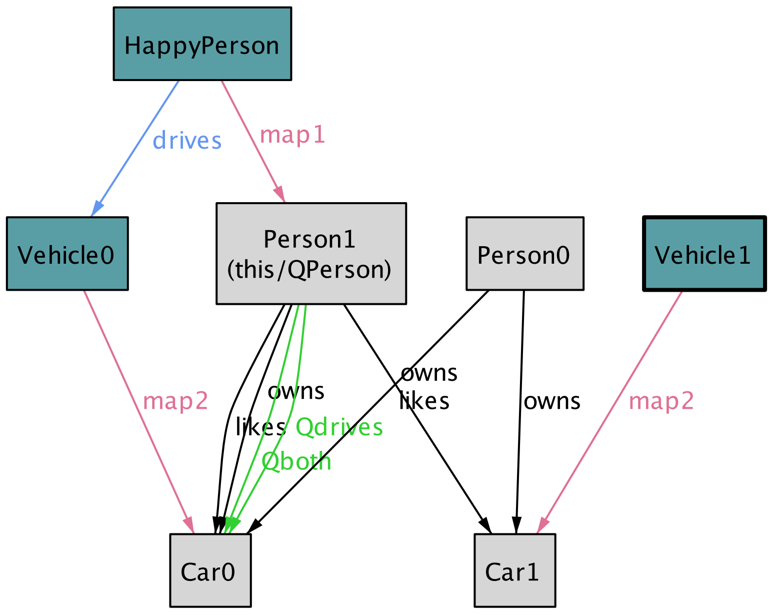


Figure 5: A counterexample generated by the Alloy analyzer for Prop. 2

Theorem 2 (Property 3 Holds). *If a happy person drives a vehicle, this person likes the corresponding car in the source model. Formally,* $\boldsymbol{Th} \models$
$(\forall h{:}\,HappyPerson, \forall v{:}\,Vehicle) v{\in}h.drives \;\Rightarrow\; map2(v){\in}map1(h).likes$

Proof. As relations *drives* and *Qdrives* are isomorphic, we need to prove the property in question for relation *Qdrives*: $\boldsymbol{Th} \models c{\in}p.Qdrives \;\Rightarrow\; c{\in}p.likes$ for all $p{\in}QPerson$, $c{\in}Car$ (recall that *QPerson* is a subset of *Person*). Now it is easy to see that the property above means that $Qdrives{\subseteq}likes$, which is obvious as $Qdrives{=}Qboth$ and $Qboth{=}owns \cap likes \subseteq likes$. □

The two proofs above show the general idea of MT analysis in QueST. The transformation definition mapping maps the target metamodel MM_T into a suitable extension of the source metamodel with derived element, $Q(MM_S)$, where Q refers to a set of queries providing the extension. When we need to prove a property that involves target metamodel elements (which is a typical case), we replace those elements by their images in $Q(MM_S)$ and rewrite the property accordingly (an accurate formal specification of this construction can be found in [DW08]). The problem thus amounts to property analysis over $Q(MM_S)$, which is a standard logical problem studied in databases: given a relational schema with constraints, MM_S, analyze the properties of the queries $Q(MM_S)$ against the schema. We plan to investigate how the results obtained by the database community can be adapted for MT in our future work.

4 Related work

Paper [ACL$^+$14] provides a useful classification of model transformation techniques. In their terminology, our present work lies in the Type 2 category which classifies Transformation Dependent and Input Independent verification techniques. More specifically, Formulating the MT as a theory is similar to *theorem proving* techniques. Calegari et. al. work [CLST11] is the closest to our work in this category. They presented encoding of ATL transformation as a logical theory in Coq: metamodels are translated to inductive types and model transformation rules to logical formulas, and properties specified as propositions in the corresponding theory; then they used Coq theorem prover to analyze the properties. The main difference of our work with [CLST11] originates from the differences in transformation components (i.e., ATL rules vs. QueST queries) which effects the shape of axioms in the theory. We briefly discussed the difference of QueST with a rule based MT language in [GDM14] from engineering perspective.

We used Alloy as the QueST underlying language, so metamodels as well as queries are encoded as Alloy specification. The authors of paper [BECG12] provide an automatic translation of ATL transformations to transformation models expressed as an OCL theory, and used OCL model finder to analyze the properties. This is similar to encoding of an MT in Alloy and using instance finder to check their properties. In [MC13], the authors used Alloy to encode QVT-R transformations and proposed an alternative enforcement semantics for that. Papers [CGR13, MRR11] proposed a translation between the UML class diagram and Alloy for the validation and verification of class diagrams. The authors in [ABK07] translate source and target metamodel to alloy and specify the transformation rules as mapping relations. In [GK15], the authors introduce a sub-language of Alloy called F-Alloy that is used for expressing a functional mapping representing a transformation. Similar to [CLST11], papers [ABK07, GK15] follow the common MT paradigm that a source-to-target transformation is specified as a set of mappings going from the source to the target elements, whereas in QueST these mappings go in the opposite direction. Finally, authors in [SLC$^+$14] employed a different technique based on symbolic execution for graph-based MT verifications.

5 Conclusion

In the present paper, we have shown how a verification of MT-related properties can be specified and checked in the QueST framework: every transformation definition in QueST amounts to its corresponding theory, and checking properties is equivalent to proving or disproving the corresponding theorems in this theory. Using a simple MT example, we demonstrated how the latter mechanism works: we encoded the theory of the example in Alloy, and used its analysis facilities to disprove a sample property. Then we provided manual mathematical proofs for the other two properties.

Using the simple example in this paper was for a demonstrative purpose. We already showed in our previous work [GDM14] that QueST is applicable to larger and more complex examples like the transformation of Class Diagrams to DB Schemas. The complexity of the example would only affect the complexity of the queries and proofs of the properties to be checked, but the proposed approach to the verification would still be valid and applicable. As a future work, we plan to apply the proposed method to a large-scale industrial application and further study how the structured nature of QueST's specification of MTs might be helpful in managing the complexity of property checking.

References

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Küster. Analysis of model transformations via alloy. In *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.

[ACL$^+$14] Moussa Amrani, Benoît Combemale, Levi Lúcio, Gehan Selim, Jürgen Dingel, Yves Le Traon, Hans Vangheluwe, and James R Cordy. Formal verification techniques for model transformations: A tridimensional classification. *Journal of Object Technology*, pages 1–43, 2014.

[BECG12] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of atl transformations using transformation models and model finders. In *Formal Methods and Software Engineering*, pages 198–213. Springer, 2012.

[CGR13] Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and uml class diagrams annotated with ocl. *Software & Systems Modeling*, 14(1):5–25, 2013.

[CLST11] Daniel Calegari, Carlos Luna, Nora Szasz, and Álvaro Tasistro. A type-theoretic framework for certified model transformations. In *Formal Methods: Foundations and Applications*, pages 112–127. Springer, 2011.

[Dis09] Zinovy Diskin. Model Synchronization: Mappings, Tiles, and Categories. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 6491 of *Lecture Notes in Computer Science*, pages 92–165. Springer, 2009.

[DW08] Zinovy Diskin and Uwe Wolter. A diagrammatic logic for object-oriented visual modeling. *Electr. Notes Theor. Comput. Sci.*, 203(6):19–41, 2008.

[GDM14] Hamid Gholizadeh, Zinovy Diskin, and Tom Maibaum. A query structured approach to model transformation. In *AMT 2014–Analysis of Model Transformations Workshop Proceedings*, page 54, 2014.

[GK15] Loïc Gammaitoni and Pierre Kelsen. F-alloy: An alloy based model transformation language. In *Theory and Practice of Model Transformations*, pages 166–180. Springer, 2015.

[Jac12] Daniel Jackson. *Software Abstractions: logic, language, and analysis.* MIT press, 2012.

[MC13] Nuno Macedo and Alcino Cunha. Implementing qvt-r bidirectional model transformations using alloy. In *Fundamental Approaches to Software Engineering*, pages 297–311. Springer, 2013.

[MRR11] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Cd2alloy: Class diagrams analysis using alloy revisited. In *Model Driven Engineering Languages and Systems*, pages 592–607. Springer, 2011.

[SLC$^+$14] Gehan MK Selim, Levi Lúcio, James R Cordy, Juergen Dingel, and Bentley J Oakes. Specification and verification of graph-based model transformation properties. In *Graph Transformation*, pages 113–129. Springer, 2014.