Towards Declarative and Incremental Model Transformation

Hamid Gholizadeh

McMaster University, Department of Computing and Software,
Ontario, Canada
mohammh@mcmaster.ca

Abstract. Model Driven Engineering (MDE) has proven to be a promising approach in software engineering. Model management and maintenance stands at the core of the MDE approach, while it still needs more theoretical and technical support for the realization of its expected functionalities, like model transformation, refactoring, migration and synchronization. In this thesis proposal, I introduce a declarative Model Transformation definition in the meta-model layer, as the foundation for model transformation. I motivate the declarative method by discussing its usefulness for incremental model transformation. This PhD proposal's objectives are the implementation of the declarative method, developing the necessary semantics for the incremental model transformation, and applying the incrementality to some MDE scenarios.

Keywords: Incremental Model Transformation, Declarative Model Transformation, Transformation Languages, Model Driven Engineering

1 Introduction

Model Driven Engineering(MDE) has proven to be a promising approach in the software development process [3]. Model Transformation(MT) means translating one model to another model and it stands at the core of the model management activities in MDE, since many operations in model management use it. There are two main MT approaches: imperative and declarative. In imperative approaches, transformation is defined by an algorithm which is traversing the source model's elements and generating the corresponding target model elements, step by step. In declarative approaches, transformation is specified in a specification language, and its imperative implementation is left to the underlying MT framework. Imperative and declarative MTs are comparable to imperative and declarative programming languages, respectively.

Since in declarative methods, users provide the MT specification, it is possible to have different implementations, verify them against the specification and optimize them, while still retaining the correctness of the implementation. Usually in declarative MT methods, since the size of MT specification decreases in comparison to its implementation, its maintainability increases. Unfortunately, it is not easy to write a declarative specifications and it needs training and practice for imperative-oriented minds to get used to this. In contrast, imperative approaches are being widely used at the moment, and many languages and tools exist for them, but they miss the advantages of the declarative approaches we just mentioned.

After transformation, it is possible that either the source model, the target model or both, change and we require the propagation of changes to make both sides consistent again. The naive way of dealing with this situation is generating the new versions of related models from scratch[1], which is usually not desirable. For example, it is obvious that the developer would not be happy if the code inside a Java function were thrown away, when only the name of that function changes in the corresponding class diagram. So we would need to have incremental transformation for these kind of situations in the sense that we only propagate changes and keep related models' elements unchanged as far as possible. To this end, declarative MT is useful, inasmuch as it provides a more abstract perspective for dealing with incrementality complexity; we will discuss this more in section 3.1.

In the next section we will introduce a declarative approach based on mathematical background, which is primarily introduced by Diskin [4]. This PhD research will implement this declarative approach on top of existing MT languages and develope the necessary semantics for incrementality. Based on this, we intend to apply incrementality to some MT scenario as a proof of concept.

2 Declarative MT : An Example

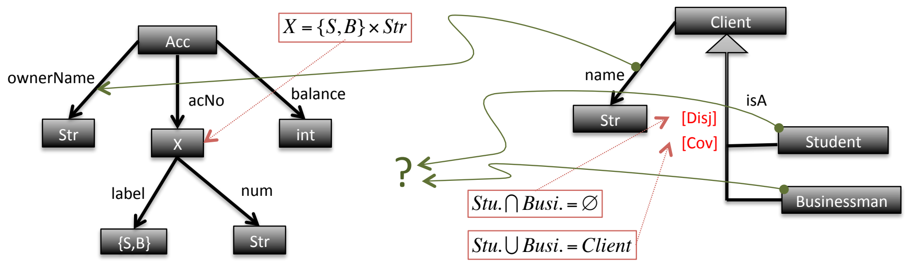
We formally represent models as Typed Graphs [5] with some constraints defined on them. A typed graph consists of three elements: *data graph*, *type graph* and *typing mapping* (a morphism from *data graph* to *type graph*). But commonly data graphs, are referred to as models, and type graphs are separately referred to as *meta-models*. For the sake of simplicity, we also stick to this naming convention.

We would like to translate account models in a bank to the corresponding client models. Account meta-model(i.e. AC) is shown in Fig. 1(a); boxes in the meta-model denote classes and primitive domains, and arrows are attributes. In our formal semantics, boxes are interpreted by sets, and attributes are functions between them. Class `Acc` has three attributes `ownerName`, `acNo`, and `balance`. The type(codomain) of attribute `acNo` is the labeled cartesian product of classes {`S,B`} (with label/field `label`) and `Str` (label/field `num`). Value `label=S` indicates that the account is a Student account, and `label=B` means it is a Business account.

Fig. 2(a) and Fig. 2(b) demonstrate the tabular and the graph representation of an AC instance, respectively. The first record in Fig. 2(a) shows a student account and the second record shows a business one.

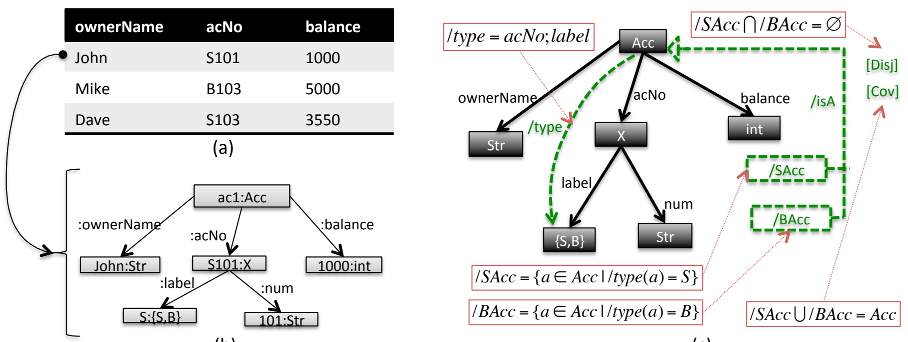
Client meta-model(i.e. CL) is shown in Fig. 1(b). The class `Client` has an attribute `name` and two subclasses(arrow with triangle head denotes subclassing):

[1] This is called batch-update



(a) AC (Account meta-model) (b) CL (Client meta-model)

Fig. 1. Account and Client meta-models



ownerName	acNo	balance
John	S101	1000
Mike	B103	5000
Dave	S103	3550

Fig. 2. (a) tabular and (b) graph representation of AC instance. (c) is augmented AC or Q(AC).

`Businessman` and `Student`. Two constraints([Disj] and [Cov]) ensure that each client is either a student or a businessman, but not both.

The informal specification of the transformation is defined as follows: each account corresponds to one client; we assumed that each client has only one account in the bank; this is implicitly stated in Fig. 1(a) by assuming that all multiplicity constraints on edges are 1..1.; the client's name corresponds to the account's ownerName, and the first component of the account number indicates the type of the client–whether he is student or businessman.

The way we approach declarative MT is by relating the elements of the target meta-model(i.e. CL) to the basic or derived elements of the source meta-model(i.e. AC). We try to do this by using an informal MT specification. In our example, we link, first, class `Client` with attribute `name` in CL to class `Acc` with attribute `ownerName` in AC. Now we would like to link classes `Stu.` and `Busi.` in CL to respective classes in AC, but there are no such classes in AC. However, we notice that these classes can be defined (derived) in meta-model AC by some suitable queries. In other words, we augment meta-model

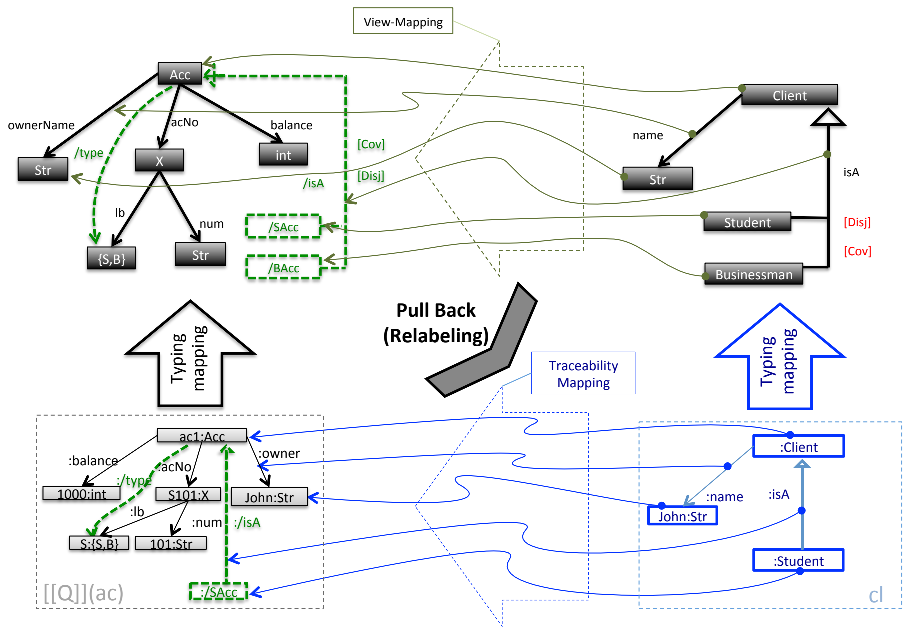


Fig. 3. Relabelling(pullback) operation executed on [[Q]](ac), which produces cl and two mappings outgoing from that: Traceability and Typing mappings

AC with new elements denoting (results of the respective) query definitions. We call this procedure *meta-model augmentation*. We augmented AC in Fig. 2(c); its augmented elements are in green dotted lines and their corresponding *queries* are attached to them in red boxes. e.g. `/type`[1] is the composition of `acNo` and `label`, and `/SAcc` represents accounts, for which their `acNo`'s first component is `S`. By having augmented AC, say Q(AC), we can continue linking elements from CL to Q(AC); e.g., we relate `Businessman` and `Student` to `/BAcc` and `/SAcc`, respectively. Complete linking from CL to Q(AC) is shown in the upper part of Fig. 3; we call this collection of links a mapping from CL to Q(AC), or a *view-mapping*, as it is analogous to the view-table mapping in database terminology.

What is described in the previous paragraph is the declarative MT definition procedure, which is a heuristic process and requires user involvement; that procedure is summarized in the more abstract view in Fig.4 by steps (1) and (2).

For a given model like ac:AC (ac of type AC), MT execution would be as follows: first, queries are executed on ac and we get an augmented source model, [[Q]](ac). Augmented ac is shown in the lower left part of the Fig.3[2]. Then, the relabelling procedure takes [[Q]](ac), Q(AC), and typing mapping from [[Q]](ac)

[1] we followed UML notation for showing derived elements by putting slash at the beginning of the derived element's name.

[2] original mode's element are in grey solid lines and augmented elements which are results of query execution, are in green dotted lines

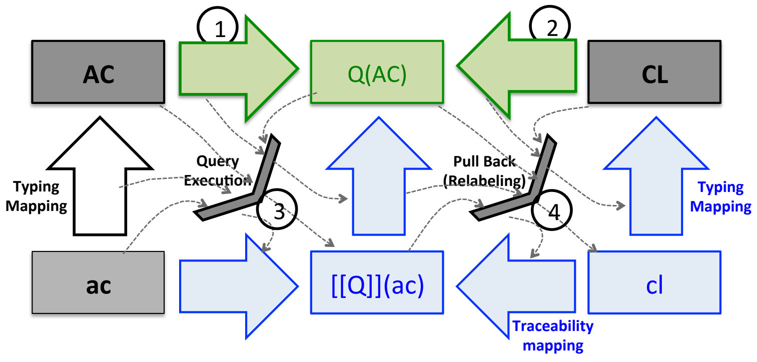


Fig. 4. *MT definition* (what users do): (1) augment meta-model by query definition, (2) define view-mapping. *MT execution*(what is done automatically): (3) query execution on model, (4) pullback operation.

to Q(AC), as well as CL and the view-mapping, and generates, cl and two mappings: a traceability mapping from cl to [[Q]](ac) and a typing mapping from cl to CL. The relabelling procedure is actually pullback operation in category theory. Fig. 3 shows in detail how the relabelling procedure acts on ac. The entire procedure described in this paragraph is abstracted in Fig. 4 and labeled by steps (3) and (4). These two steps can be implemented automatically and don't require user involvement.

3 PhD Proposal

In this section we discuss the incrementality issues and why we think declarative MT is useful for dealing with them. Then, we list the PhD research objectives and explain them.

3.1 Declarative MT and Incrementality

In this section we argue that declarative MT makes incremental MT easier. Suppose that we run MT, and the target model, T_M, is generated from the source model, S_M. Later S_M changes and we want to reflect this change on T_M. One naive approach is to regenerate T_M from scratch (so called batch-update). This approach has two main disadvantages: first, when models are huge, and changes happen frequently, running the transformation for every change is very costly in time; second, it might happen that T_M is also already changed —without loosing its consistency with S_M– and we want to respect the T_M changes, but batch-update completely ignores the T_M changes. So, it is better to have an incremental MT, in the sense that the MT engine propagates only the changes of S_M to T_M. To have incrementality, we need to deal with some issues: *private* and *shared* parts of models, *minimal change* on the target, and *conflict resolution.*

We discuss them briefly in the following paragraphs and mention how declarative MT may help.

***Private* and *shared* area.** Sometimes, when S_M changes, it is not necessary to change T_M, to make the two models consistent again. In other words, the changed S_M still remains consistent with T_M. In this case we say that the changes happened in the *private* area of the source. In contrast, if the changes happened on the part which caused S_M to lose its consistency with T_M, then we say that the changes happened in the *shared* area of S_M. Distinguishing between *private* and *shared* areas obviate the need to run the transformation every time that some changes happen on the *private* area. Defining the transformation on the meta-model layer –as we proposed in the previous section– makes it easy to identify *private* and *shared* areas; e.g., it is completely clear from our MT definition in Fig.3 that `balance` properties of AC instances belong to their *private* areas, since if the balance of an account changes, we do not expect to change anything in the corresponding client model.

Minimal change. As we said, when T_M changes, we would like to propagate it to S_M; the problem is to find a suitable translation on S_M; there might be more than one way to do this translation. When T_M is a view of S_M, this situation is analogous to a well-known problem in the database community called the view-update problem. In the database context, the Constant Complement(CC) approach [2] suggests that if we fix the complement of the view, translation of the view updates can be defined uniquely. Choice of view complement can be interpreted as the definition of an update policy. In general, we are interested in choosing a translator of updates in a way that imposes *minimal* change on S_M; that is changing S_M as minimally as possible; the best case is not changing it at all. We need to formally define this minimality and its semantics. Recently, some work has been done on the view-update problem, extending the CC approach mentioned above using category theory [9]. We believe that the categorical approach to MT definition in meta-model, provides a basis for extending the CC approach to the MT world.

Conflict resolution. Sometimes S_M and T_M evolve at the same time; e.g., software team members might work separately on some class diagrams and sequence diagrams, or they might work on code and class diagrams simultaneously. Updated models might be in conflict, and in most cases user involvement and a heuristic approaches are necessary to resolve the conflicts. A conflict resolution objective is to support users in resolving the conflicts . Looking at MT at a more abstract level provides a better understanding of the MT behaviour and helps to develop supporting tools and their semantics for conflict resolution purposes.

3.2 PhD work definition and challenges

The following list summarizes intended PhD goals:

1. *Graphical language* design and implementation, which provides a GUI for declarative MT definition
2. Adapting existing *query languages* for meta-model augmentation.

3. Implementing a *transformation engine*, using existing MT languages.
4. Defining necessary *semantics for incrementality* based on the introduced declarative approach.
5. Applying incrementally to some MT scenarios.

We intend to implement the declarative MT inside the eclipse framework as it provides some good features like Ecore tools and APIs[13]. For designing the graphical language, we will define its meta-model and provide an editor for the users to define model transformations. It would let users to define the augmentation of source meta-models and the view-mappings.

OCL[12] at the moment is used as a side-effect free query language for specifying constraints on UML diagrams. Besides that, EOL[11] as the core language of the Epsilon framework provides a query definition facilities. We might choose one of them to integrate as *query language* in our implementation, since there is already good tooling support for them.

Transformation engine of the framework would be responsible for executing transformation defined by user in graphical GUI. the engine is responsible for query execution on given models. Further, it implements relabelling operation and building the traceability and typing mappings from generated target model, to respective source model, and target meta-model.

Finally we will develop a precise semantics for supporting *incrementality* based on the introduced declarative MT approach, and apply it to some MT scenarios as proof of the concept.

4 Related works

For each imperative and declarative MT approaches, there exist some languages and tool support. Epsilon framework[11] provides a set of imperative languages and tools for model management tasks. Story Diagrams(SD)[6] combined graph grammars —which are means to define transformation constructively– and activity diagrams to provide a mechanism for specifying model transformations. ATL [10] and Viatra2 [1] are both providing imperative and declarative transformation languages. Our introduced approach provides a declarative structure(meta-model augmentation, and view mapping) which is implicit and entangled in the current imperative MT approaches.

Triple Graph Grammars(TGGs)[7] are extension of regular graph grammars and used for specifying the correspondence between the S_M and T_M elements declaratively. TGG approach is different from our declarative approach in the sense that in TGG, the MT definition between S_M and T_M is specified constructively between model elements, while in our declarative approach, MT is completely specified in the meta-model layer between the meta-model elements. So we raised the level of abstraction in relation definition between source and target.

QVT-R is declarative language of QVT [12] for model transformation. It has some semantic issues [14] and is not actively used in industry. There exists many similarities between the QVT and the TGG concepts[8] and like TGG, relation definition between S_M and T_M is not in the meta-model layer.

5 Conclusion

We introduced declarative MT by an example, and motivated its implementation by discussing its usefulness for incrementality. PhD objectives are implementing declarative MT, developing precise semantics for incrementality based on introduced declarative method, and applying incremental method to some MT scenarios.

References

1. András Balogh and Dániel Varró. Advanced model transformation language constructs in the viatra2 framework. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287. ACM, 2006.
2. François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, 1981.
3. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.
4. Zinovy Diskin. Model synchronization: mappings, tiles, and categories. In *Generative and Transformational Techniques in Software Engineering III*, pages 92–165. Springer, 2011.
5. Hartmut Ehrig and Karsten Ehrig. Overview of formal concepts for model transformations based on typed attributed graph transformation. *Electron. Notes Theor. Comput. Sci.*, 152:3–22, March 2006.
6. Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In *Theory and Application of Graph Transformations*, pages 296–309. Springer, 2000.
7. Ulrike Golas, Leen Lambers, Hartmut Ehrig, and Holger Giese. Toward bridging the gap between formal foundations and current practice for triple graph grammars. In *Graph Transformations*, pages 141–155. Springer, 2012.
8. Joel Greenyer and Ekkart Kindler. Reconciling tggs with qvt. In *Model Driven Engineering Languages and Systems*, pages 16–30. Springer, 2007.
9. Michael Johnson and Robert Rosebrugh. Constant complements, reversibility and universal view updates. In *Algebraic Methodology and Software Technology*, pages 238–252. Springer, 2008.
10. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.
11. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Eclipse development tools for epsilon. In *In Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.
12. OMG, `http://http://www.omg.org/spec/`. *OMG Specifications*, 2013.
13. David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.
14. Perdita Stevens. Bidirectional model transformations in qvt: semantic issues and open questions. *Software and Systems Modeling*, 9:7–20, 2010.