Breaking Up the Transformation Chain

Bert Vanhooff
Katholieke Universiteit Leuven
Celestijnenlaan 200A
Leuven, Belgium
bert.vanhooff@cs.kuleuven.be

Yolande Berbers
Katholieke Universiteit Leuven
Celestijnenlaan 200A
Leuven, Belgium
yolande.berbers@cs.kuleuven.be

ABSTRACT
Both Model-Driven Software Development (MDSD) and Model Driven Architecture (MDA) emphasize the importance of precise machine-readable models and automatic transformations on these models. In this paper we identify the need to externally specify transformation units in terms of required and provided model properties. We also present shortly how one can use semantically rich transformation traceability information as a specific kind of externally quantifiable property. Using precise specifications allows to break up monolithic transformation implementations into modular chains of transformation units that only depend each other's, externally specified, output model characteristics.

1. INTRODUCTION
The Model-Driven Software Development (MDSD) [2] paradigm offers a methodology for the model-based development of software. It can be seen as an addition to the Model Driven Architecture (MDA) [7] of the OMG. The latter focuses more on providing standardized technology specifications for notation and tool-interpretability. The OMG also provides a vision document but this is too generic to be called a methodology. Both MDA and MDSD emphasize the use of

1. Metamodeling for modeling-language design
2. Precise models
3. Automatic transformations

While the focus of MDSD is more on metamodeling (1) of domain-specific languages, MDA pushes the use of the Unified Modeling Language (UML) (2) [8], both not excluding the other. UML offers metamodeling facilities of its own through the profiling mechanism. This mechanism is limited to extending existing UML elements (keeping their inherent capabilities and limitations) with new semantic meanings. Applying profiles allows to tailor the UML to a specific domain and has the advantage of staying within a well-known language for which there is existing tool support. The techniques discussed in this paper are primarily targeted towards the use of UML models and UML-to-UML transformations, heavily using the profiling mechanism.

Since the UML is not conceived to be an execution platform, models written in this language can not be run directly (although there are some exceptions, xUML [10]). They need to be translated to code for a concrete platform (e.g. Java) in order to be useful. How such a translation should be done is specified in model transformation specifications. These specifications contain precise rules, describing how to relate source model elements to target model elements (Query View Transform [5], [9]). A model transformation is seldom a monolithic block but is rather composed of smaller, what we like to call, chained *transformation units*. We define transformation units as reusable building blocks of transformations. More information on transformations is given in section 2.

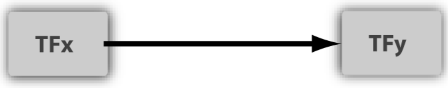
This paper deals with breaking up large monolithic transformations into smaller units that are more easily definable, reusable, adaptable, etc. It is easier for these small units to support the agile characteristics of MDSD such as small team development, timeboxed iterations and iterative dual track development [2]. We propose to break up transformations into smaller units that cooperate by explicitly depending on each other's output model characteristics. We will address these dependencies in general and specifically we will introduce the notion of transformation traceability as a special kind of externally quantifiable model property. Finally we will give a a complete example of breaking up a transformation, allowing the reader the get a grasp of the advantages of compact transformation units. We will wrap up this paper by presenting some related work and drawing conclusions.

2. TRANSFORMATIONS
The ultimate goal of the MDA is to start from highly abstract models and gradually move to more concrete models in order to eventually end up with models that can straightforwardly be implemented on a concrete technology platform. In the ideal case a complete model transformation is accomplished using a chain of small and reusable (possibly off-the-shelf) transformation units. Each transformation unit applies well-defined and limited changes to a model.

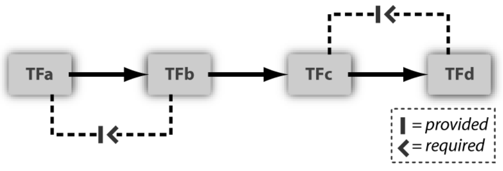
In this section we first discuss how transformation units can depend on each other, followed by a discussion of the concept of transformation traceability. We end the section with a presentation of a transformation traceability profile.

2.1 Dependable Transformation Chain

We argue that it can be beneficial to denote precise dependencies between subsequent transformations units. Figure 1 shows two versions of the same transformation. Full lines indicate transformation unit execution sequence, while dotted lines indicate dependencies. The transformation (a) is composed out of two other independent transformations — the order of their application does not matter. They both have to apply a lot of changes to the model in order to adhere to their requirements. As a consequence they are not so easy to implement. To ease the implementation effort we can try to decompose these large units into smaller chunks. The second version (b) shows the results of a possible decomposition. There now are four smaller transformation units, from which transformation unit TFb relies on some model properties that are delivered (guaranteed) by TFa. TFd and TFc are related in a similar manner. One can immediately see that the transformation units of the second transformation are much smaller and should be easier to create and to maintain — at the cost of added dependencies of course.



(a) Independent – course-grained – chain



(b) Dependable – fine-grained – chain

Figure 1: Conceptual transformation chain

Good practice in software engineering shows us that it is useful to separate specification (what it does) from implementation (how it does it). This is exactly what we have to do in the area of transformations. Different transformation units need to explicitly define their dependencies at a high enough level so that transformation units do not rely on each other's implicit low level implementation details. By this we mean that they should only rely on externally quantifiable properties of the model that are present after the execution of the previous transformation unit. This implies that each transformation unit should clearly define what one can expect (require) from its provided output model and facilitates loosely coupled transformation units. Such a philosophy is comparable to component based software engineering, which promotes the use of software units that only have explicitly defined dependencies in terms of provided and required interfaces.

UML profiles seem to be a good way to formally specify in- and output model characteristics, and as consequence dependencies between transformation units. They can be used to constrain a model in almost any possible way, so when a profile is applied one can be certain of some specific model properties. Examples of dependencies described in terms of profiles can be as simple as "TFb requires the CORBA profile [6] to be applied". It is also possible to specify dependencies at an even higher level, for example "TFd requires the presence of interfaces for each class", which can be fulfilled by a number of profiles. In this paper we will specifically talk about dependencies on transformation traceability information: "Tfb requires traceability links of type X to be present". More investigation on other types of dependencies are deferred to future work.

None of the dependency examples, given in the previous paragraph, directly refer to other transformation units. They rather describe dependencies at a higher level. The examples are all in the form of "TFx requires Y", with Y some model characteristic. This indicates that the opposite relation should also exist: "TFy provides X". A require/provides couple can than realize a dependency relationship — indicated in the figure using symbols for required and provided features. The terminology we use here is deliberately chosen to match that of component based software engineering.

In the following section we will give a concrete example of how subsequent transformation units can use each other's transformation traceability links. We will also argue that sometimes it is even required that transformation units are aware of each other's manipulations of the model.

2.2 Transformation Traceability Dependencies

In order to explain what we understand by transformation traceability dependencies, we will introduce an example transformation chain. The transformation units in this chain will make use of profile applications to guide their transformation process. The chain is constructed of two transformation units (rounded rectangles in figure 2). The first one (*GetSetTF*) adds simple get- and set-operations to a class for each of its appropriately marked attributes. The following transformation unit (*LogTF*) adds logging logic. Each time an attribute value is changed, this change will be logged.

An attribute value should only be changed through its corresponding set-operation. Since all set-operations are generated by the (*GetSetTF*) transformation unit we can enforce this behavior. The *LogTF* transformation unit can than rely on that set-operation to detect an attribute value change. This creates an obvious dependency between the two units: *LogTF* can add a call to its logging operation to each generated set-operation. To allow *LogTF* to do this, (*GetSetTF*) has to leave behind enough information in order to find set-operations. Figure 2 shows the effects of the two transformation units on a single class named *Car*. After the *GetSetTF* transformation unit, generated get- and set-operations are added to the class. Traceability links are added that indicate the relation between each get- or set-operation and its corresponding attribute. Next, the *LogTF* transformation unit adds its *LogItem()* operation to the class. From the traceability links left behind by *GetSetTF* it can derive which operations modify the value of an attribute, so it can add the necessary operation calls. In turn it leaves behind

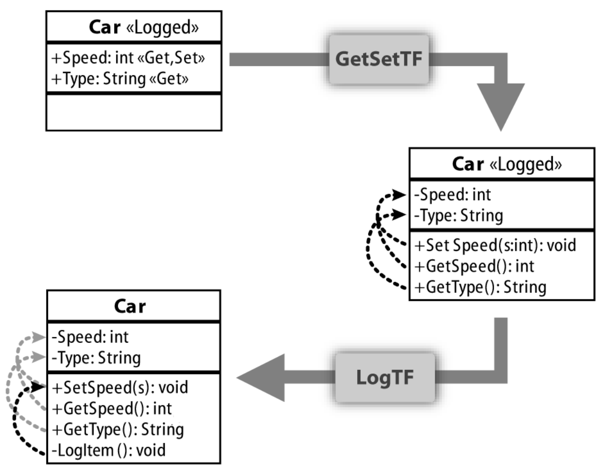


Figure 2: Two traceability-aware transformation units executed on a single class.

a traceability link that indicates when an item is logged — whenever a set-operation is called in this simple case.

We must notice that, in the case of the *GetSetTF* it is possible to locate each operation that modifies an attribute value from the actions (from the UML action semantics) that belong to that operation, without needing the transformation traceability links. This is in the assumption that the model is complete; even than these explicit traceability links offer the advantage of easier transformation implementations because extensive searching is avoided.

This example perfectly illustrates why transformations are sometimes required to be aware of each other. Take for example the *Speed* attribute. It is annotated as *Get*, *Set*. If the *GetSetTF* and the *LogTF* transformation units were unaware of each other, they would both have to generate a set-operation for the attribute. The *GetSetTF* has to generate it because the *Set* stereotype directly says so. *LogTF* has to generate it in order to have a single modification point of the value of *Speed*. Two versions of the set-operation will lead to unexpected behavior at execution time.

We showed the advantages offered by using transformation traceability links. The next section introduces a profile that allows to insert traceability links into UML models.

2.3 Transformation Traceability Profile

We defined a profile (*TFTraceProfile*) that introduces transformation traceability concepts into the UML. We will only discuss this profile superficially — for further detail we refer to [11]. Basically the profile subclasses the UML *Dependency* class to realize traceability links and to relate each traceability link to a transformation unit. Because we do not want any (historical) information to get lost we only allow obsolete elements to be marked as such instead of being deleted for good.

Of course just inserting traceability links between each (group of) source and target element(s) does not add much value to our models. Doing so creates an abundance of less useful links that just say "target *is derived from* source". While this can be very useful for the internals of transformation engines (see section), it is not the desired effect for our situation. That is why we require two additional actions:

1. Definition of sub-profiles that add rich semantics to the basic traceability links for each sub-transformation.
2. Insertion of traceability links by the transformation units themselves; they know best where and which links to add.

There are two possibilities to extend the traceability profile. We can define a completely new, transformation unit specific, profile based on *TFTraceProfile*. This derived profile will than only contain new types of traceability links with a specific semantic meaning. Alternatively we can add traceability links to existing profiles by creating a new profile that imports both the elements of the existing profile and those of *TFTraceProfile*. It makes sense to extend an existing profile with the traceability profile if it is applied to a model as part of the result of a transformation.

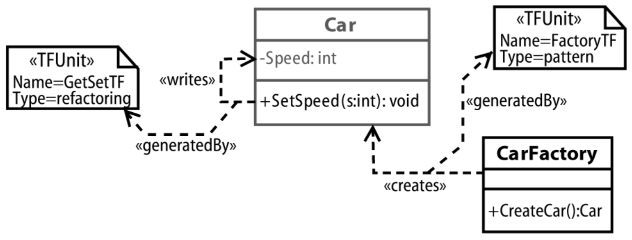


Figure 3: Example: Use of the traceability profile.

Figure 3 gives an example of the application of (two extensions of) our traceability profile. We can see that two transformation units have changed the model. Both have added their own semantically rich traceability links. The *GetSetTF* has added the *writes* link while the *FactoryTF* has added the *creates* link. Using these links, a subsequent transformation unit can precisely determine how generated elements are related. This makes richer specification of transformation units possible.

3. A COMPLETE BREAK UP

In this section we give an example of how a monolithic, non-flexible, transformation can be decomposed into a flexible chain of dependent transformation units.

Figure 4 denotes the starting situation. It shows a monolithic transformation unit *PersistTF*, which is responsible to add automatic persistency features to a model. The modeler only needs to indicate which attributes need to be persistent and which attributes trigger a save operation when their value is changed. Such annotations can be made using a profile — *PersistencyProfile* in this case. *PersistTF* generates three things: unique ID attributes for each class (as its primary key), set-operations for each attribute that is marked as a save-trigger and the persistency logic itself. It is required that the *PersistencyProfile* has been applied to the model before execution of *PersistTF*. Because this

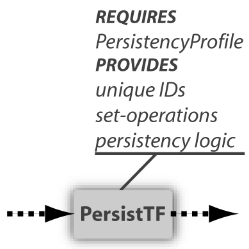


Figure 4: A monolithic transformation.

transformation unit is fairly large it can be difficult to make changes to it, digging in the transformation code. For this and other reasons, mentioned throughout the text, we will decompose this unit into smaller units using the dependable transformation chain approach.

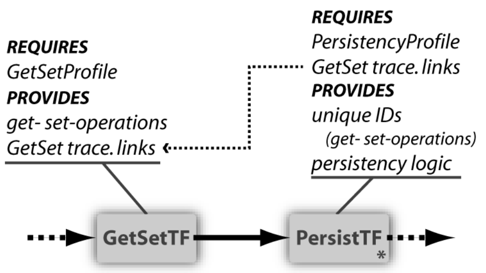


Figure 5: Adding dedicated set- and get-operation generation.

To decompose *PersistTF*, we have to split off every functionality that is not a core persistency issue or any functionality that can potentially be useful for other transformation units. We start by relieving *PersistTF* from having to generate set-operations. This will be made the responsibility of the *GetSetTF* (see figure 5). This unit requires the *GetSetProfile*, which defines stereotypes to mark attributes with get or set, to be applied. Obviously it generates get- and set-operations accordingly. For *PersistTF* to be able to use these operations correctly it requires *GetSetTF* to provide the necessary traceability links. In fact *GetSetTF* does not have to provide them but they have to be present and it seems that *GetSetTF* is the best candidate to do so.

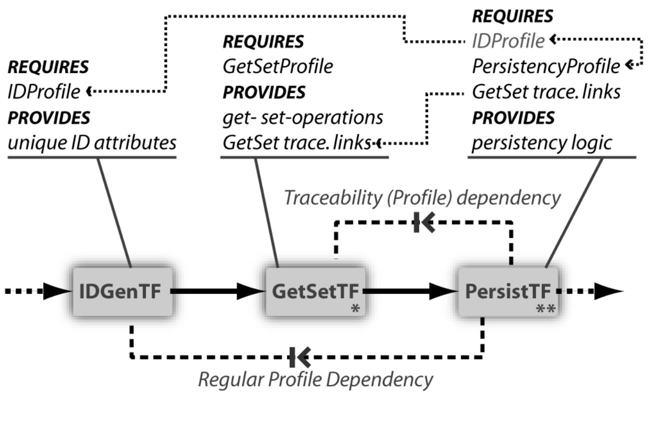


Figure 6: Adding dedicated ID attribute generation.

Adding the *GetSetTF* unit does not relieve *PersistTF* from having to generate set-operations completely. It still has to generate ID attributes and, accordingly, accompanying set-operations since these attributes are not known at the time that *GetSetTF* executes. To solve this we introduce yet another transformation unit, called *IDGenTF* (see figure 6). This unit will only generate ID attributes for classes. Adding this unit to the front of the chain enables *GetSetTF* to generate get- and set-operations for the newly added ID attributes. *PersistTF* is now relieved from generating IDs but it still needs to know which attributes are the IDs, so it requires *IDGenTF* to leave behind this information. This can be done by leaving the *IDProfile* marks (partly) in place. A curious phenomenon arises. The dependency from *PersistTF* on *IDProfile* has the interesting property that *PersistencyProfile* is actually an extension of *IDProfile*. So, it is sufficient that *PersistTF* only requires *PersistencyProfile*.

4. RELATED WORK

We proposed to define dependencies between transformation units by means of profiles that encapsulate model properties in order to modularize transformations. In order to better support breaking up transformations into small units transformation traceability information was introduced. This brings us to two areas of related work, namely requirements traceability in UML and transformation languages with respect to their facilities for reuse and transformation traceability.

Requirements traceability is used particularly in software evolution management. When a stakeholder issues a new or changed requirement we want to be able to easily localize the (modeling) artifacts that are influenced by that requirement. Managing the links between (textual) requirements and implementing artifacts at several levels throughout the development process is generally referred to as requirements traceability. A reference metamodel for representing requirements traceability links within the UML is described in [4]. The authors of that paper choose to integrate their mechanisms into the UML in order to represent all software artifacts in one model. We believe that it can beneficial to integrate our approach towards transformation traceability with an UML-based requirements traceability approach since there is some overlap between the two.

Currently many model transformation languages and tools are available. Many of them are just in an experimental phase ad offer very little tool support. We shortly discuss the IBM Model Transformation Framework [3], the Atlas Transformation Language [1] and the latest Query View Transformation (QVT) proposal issued by the QVT-Merge group [9]. Both the MTF and the ATL are more or less inspired by earlier QVT proposals. The MTF is the simplest language, offering only declarative language constructs. Both the ATL and QVT are hybrid languages. All three languages automatically generate traceability links between every source/target model element related by the transformation. These links are typed by their creating transformation rule and cannot be manually created by the user. In the MTF these links are not user-accessible at all. The ATL only offers the ability to export generated links as a separate model. QVT offers some operations in order to use the links in the transformation itself but only in an indirect fashion, for example to search all generated target elements that are generated from one source element. In contrast, our approach types the links by stereotypes with arbitrary semantics, which are defined externally and independently of any transformation language. These links are completely controlled by the implementer of the transformation. In summary, the traceability support as is provided by the current transformation languages lacks flexibility in terms of the possibility to insert and access custom traceability links programmatically. To our knowledge only QVT has language support for reusing externally defined transformations to construct a transformation chain. However, it is not clear if traceability links remain available between subsequent transformations. External specification of transformations, describing expected and provided model properties, is not offered by any of the transformation languages.

5. CONCLUSIONS AND FUTURE WORK

We stated that there is a clear need to specify the effects of transformation units independently from their implementation in order to make them more manageable. UML profiles are a good candidate to specify these effects formally and explicitly. In this way, a transformation chain can be compared to a component composition. We also introduced the notion of semantically rich transformation traceability as one way in which to break up a large transformation into a chain of smaller transformation units. Transformation traceability information is one kind of information that can be required or provided by a transformation unit and provides an extra opportunity to better modularize transformations. We defined a profile in order to insert traceability constructs.

Specifying the required and provided model characteristics in formal manner contributes to the agile aspect of MDSD because only than can several transformation units be developed independently. Units can be swapped in and out of the transformation chain in a controlled fashion, contributing to the development of product lines.

Our future work includes the refinement of our traceability profile, adding as much generally useable constructs as possible. Furthermore we are going to investigate more thoroughly in which ways profiles can be used to specify a transformation unit's required and provide model features.

6. REFERENCES

[1] Atl : Atlas transformation language, 2002. http://www.sciences.univ-nantes.fr/lina/atl.

[2] J. Bettin. MDA Journal - Model-Driven Software Development. april 2004.

[3] IBM Alphaworks. Model transformation framework, 2004. http://www.alphaworks.ibm.com/tech/mtf.

[4] P. Letelier. A Framework for Requirements Traceability in UML-based Projects. 2002.

[5] Object Management Group. Query/view/transformation rfp, 2002. http://www.omg.org/cgi-bin/doc?ad/2002-4-10.

[6] Object Management Group. UML Profile for CORBA, v 1.0, 2002. www.omg.org/cgi-bin/doc?formal/02-04-01.

[7] Object Management Group. MDA Guide Version 1.0.1, 2003. www.omg.org/cgi-bin/doc?omg/03-06-01.

[8] Object Management Group. UML 2.0 Superstructure FTF convenience document, 2004. www.omg.org/cgi-bin/doc?ptc/2004-10-02.

[9] Object Management Group. Qvt-merge group submission for mof 2.0 query/view/transformation, 2005. http://www.omg.org/cgi-bin/apps/doc?ad/2005-03-02.

[10] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[11] B. Vanhooff and Y. Berbers. Supporting Modular Transformation Units with Precise Transformation Traceability Metadata. august 2005.