

# UniTI: A Unified Transformation Infrastructure <sup>\*</sup>

Bert Vanhooft, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers

Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium  
{bert.vanhooft,dhouha.ayed,stefan.vanbaelen,wouter.joosen,yolande.berbers}@cs.kuleuven.be

**Abstract.** A model transformation can be decomposed into a sequence of subtransformations, i.e. a transformation chain, each addressing a limited set of concerns. However, with current transformation technologies it is hard to (re)use and compose subtransformations without being very familiar with their implementation details. Furthermore, the difficulty of combining different transformation technologies often thwarts choosing the most appropriate technology for each subtransformation. In this paper we propose a model-based approach to reuse and compose subtransformations in a technology-independent fashion. This is accomplished by developing a unified representation of transformations and facilitating detailed transformation specifications. We have implemented our approach in a tool called UniTI, which also provides a transformation chain editor. We have evaluated our approach by comparing it to alternative approaches.

## 1 Introduction

Model transformations are a key ingredient of Model Driven Development (MDD). They can for example be used to add details to a model, incorporate non-functional concerns, convert between different types of models, and refactor certain constructs within a model. Model transformations quickly become complex when they need to address many concerns at once. Monolithic transformations, as most non-modularized software entities [1], have some inherent problems: little reuse opportunities, bad scalability, bad separation-of-concerns, sensitivity to requirement changes, etc. A number of these problems can be solved by decomposing a transformation into a sequence of smaller subtransformations, i.e. a transformation chain.

Currently, most transformation technologies do not very well support reuse and composition of subtransformations as high-level building blocks. One of the causes is the fuzzy distinction between specification, implementation and execution of transformations [2]. We define these terms as follows:

**Implementation** is the transformation source code, as seen by the developer. For many common transformation languages this comes down to a set of mapping rules.

**Specification** is the documentation that describes how a transformation behaves, independently of its concrete implementation. We focus on the functional interface (similar to programming languages) of a transformation in terms of input and output model types.

---

<sup>\*</sup> The described work is part of the EUREKA-ITEA MARTES project, and partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders).

**Execution** is the runtime instance of a transformation that has concrete input models and produces one or more output models.

Furthermore, current transformation technologies offer little to no support for combining transformation technologies even though in practice different technologies may be suitable for different parts of a transformation chain. Lastly, the focus of transformation technologies on local implementations prevents them from addressing issues that go beyond the boundary of a single transformation such as end-to-end traceability.

In this paper we propose a technology-neutral view on model transformations: the Unified Transformation Representation (UTR). We have implemented a tool called UniTI (Unified Transformation Infrastructure) based on UTR that facilitates transformation composition and execution without having to know underlying implementation details. UniTI provides the groundwork to incorporate cross-transformation services such as traceability, mentioned in the previous paragraph.

The remainder of this paper is structured as follows. In Section 2 we give a concise overview of current transformation techniques and identify a number of concrete problems concerning (re)usability and composability. We then list a set of characteristics that a solution to these problems must possess (Section 3). In Section 4 we present the Unified Transformation Representation; the implementation of the latter – UniTI – is discussed in Section 5. We summarize related work in Section 6 and evaluate our approach by comparing it to alternative solutions in Section 7. Finally, we round up by drawing conclusions and identifying future work in Section 8.

## 2 Using and Composing Transformations

Current transformations technologies all have their own specific vision on model transformations. In Section 2.1 we give a concise overview of a selection of available transformation technologies. We then zoom in on a number of shortcomings of transformation technologies related to the creation of transformation chains in Section 2.2.

### 2.1 Characteristics of Current Transformation Technologies

A lot of effort has already been spent on the development of suitable languages to express model transformations. Key examples are ATL [3], MTF [4], VIATRA [5], UMLAUT [6] and OMG's QVT [7]. Note that generic programming languages such as JAVA can also be used to implement transformations. The main objective of specialized languages is to provide a number of powerful yet easy to use constructs to express relations between model elements. Many of the mentioned transformation languages take a different approach to implementing transformations. They differ in notational style (graphical:VIATRA; textual: ATL, MTF or a combination:QVT), specification style (imperative:JAVA; declarative:MTF or hybrid:ATL, QVT), directionality (unidirectional:ATL or omnidirectional:MTF), supported model types (UML [8]:UMLAUT; MOF [9]; Ecore [10]), etc. The examples given here cover only a small subset of available transformation languages; a more complete discussion of these and other differences between transformation languages can be found in [11].

In this subsection we describe to what extent one must be familiar with a transformation's implementation in order to use it in a meaningful way. We take ATL, MTF and JAVA as reference languages; we believe these cover a substantial part of the existing transformation approaches.

MTF enables the implementation of transformations by specifying an arbitrary number of declarative mapping rules. Each one of these rules denotes a relationship between one or more model elements and can call out to other rules. An MTF transformation is always omnidirectional, which means that any of the (meta)models involved can be designated both as input or as output. Moreover, a transformation can be initiated from any of the mapping rules, possibly producing a different result. In order to execute an MTF transformation we need to select an appropriate *initial rule* and select the *direction* of the transformation. One should be very careful when choosing a direction. It is not because a transformation turns model A into B that applying the same transformation to B yields A. We will not go into the intricate details of omnidirectional transformations but we emphasize that it is far from trivial to choose a specific direction and initial rule without being very familiar with its implementation details.

In ATL we use transformation rules that have both declarative and imperative characteristics to relate a number of models. The transformations implemented as such are unidirectional. ATL transformations are, to some extent, metamodel independent – references to metamodels are not included in the implementation. Only when we execute a transformation, we need to select concrete metamodels. The range of valid metamodels directly depends on the set of transformation rules. For example, if rules only refer to 'Class' and 'StructuralProperty' we can execute the transformation for any version of the UML metamodel. When we add an additional rule that refers to 'Port', we can only choose UML2 since that element is not present in lower versions. Hence, the set of valid metamodels can only be derived by studying the implementation.

A transformation written in JAVA, or any other general purpose language, accesses models directly via the model repository. Hence, the developer is not bound to any rules to implement a transformation. There is no standard technique to specify the input and output models; this can be done through method parameters, command-line arguments, files, etc. The same goes for specifying the entry point of the implementation (which method should be invoked?). So, in case of JAVA, we need complete knowledge of the implementation in order to use it.

## 2.2 Limitations

In this subsection we identify a number of deficiencies in the area of current transformation technologies with respect to using transformations as (reusable) building blocks in transformation chains.

**Incomplete Specification** It is clear from the previous subsection that an intimate knowledge of a transformation's implementation is required in order to use it. The following elements are not clear from the implementation alone:

- MTF : direction; initial rule
- ATL : concrete metamodels

- JAVA : concrete metamodels; direction; entry point, ...

In order to be able to (re)use, compose and execute transformations we must clearly fix the possible values of these enumerated properties.

**Imprecise Specification** When we have filled the specification gaps of each technology, we can say that a transformation is mainly characterized by its input and output model types. Metamodels are used to type models: a model *conforms to* a metamodel. This implies that if we know the metamodel we automatically know what kind of models to expect. This is indeed true, but the sheer amount of possible models that conform to a (complex) metamodel often make this an inadequate way to type models (from a transformation point of view). We give examples in three categories:

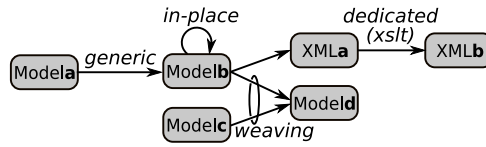
- *Metamodel variation* In order to allow a greater flexibility, a metamodel can deliberately be left incomplete. For example, the UML has a number of so-called semantic variation points, which need to be resolved before using the metamodel.
- *Metamodel delimitation* Complex metamodels such as the UML are not often used completely; a transformation usually only considers a subset of UML. For example, a UML to RDB (Relational DataBase) transformation, only takes structural elements such as classes and associations into account and ignores other elements such as actions and states.
- *Model structure* Next to global metamodel concerns, transformations may also make assumptions at the model level. For example, requiring specific model elements such as a ‘Car’ class, a directed association named ‘fuel’, etc.

Mind that the above issues are not due to shortcomings of particular metamodeling languages. Metamodels such as the UML are conceived to accommodate a wide range of modeling possibilities, making them reusable in many domains. A transformation, however, often only makes sense on a small subset of all the possible instances of a metamodel.

Current transformation languages cannot directly express such subsets. The best they can do is check whether a model belongs to the expected subset at execution time, which requires additional code that pollutes the implementation. This also means that the requirements on the involved models are hidden inside the implementation. In order to check whether we can successfully execute a transformation or whether transformations can be connected in a chain, these subsets must explicitly be defined as part of its specification.

**Technology Lock-in** Different technologies can be suited to implement different types of transformations. Figure 1 suggests a technology for each subtransformation in an imaginary transformation chain.

To go from *Modela* to *Modelb* we use a generic transformation language that supports any metamodel (e.g. ATL). To make minor changes to *Modelb* we prefer an in-place (source=target) technology (e.g. JAVA). In order to combine *Modelb* and *Modelc* we use a model weaving engine (e.g. AMW [12]). Finally, we use a domain specific transformation language (XSTL) to transform XML models.



**Fig. 1.** Different types of transformations in a transformation chain.

If we choose a different technology for each subtransformation, we must ensure the ability to combine the technologies. Since current transformation technologies offer no (or only very basic by QVT) possibilities to call out to other independently defined transformations, it is very hard to mix different technologies. In order to realize a cross-technology transformation composition we need to write additional, external *glue code*.

**Megamodel Concerns** Megamodel management is the term introduced by Bézivin et al [12] to indicate the need to establish and use global relations between macroscopic MDD entities such as (meta)models and transformations while ignoring the internal details of these entities. Reusing and composing transformations in transformation chains is one of the use cases within this area.

Current transformation languages focus on implementation details of individual transformations and cannot express much at the level of complete models or transformations. A higher level transformation infrastructure is needed to offer additional functionality on top of a transformation chain, such as end-to-end traceability [13].

### 3 Characteristics of Reusable & Composable Transformations

Since we are using transformations as building blocks, our point of view is related to the field of component based software engineering (CBSE) [14]. We will use the main principles of CBSE as guidelines to solve the issues raised in the previous section. These principles, reformulated to fit the transformation domain, are summarized below. We also describe deficiencies of current transformation technologies with respect to each principle.

**Black-Box principle** *The black-box principle is the strict separation of a transformation's public behavior and its internal implementation. Implementation hiding makes any technique an eligible candidate to implement the transformation.*

As discussed in the previous section, transformation specifications in the considered languages are both incomplete and imprecise. We need to look into their implementation in order to use them, which violates the black-box principle.

**Composition** *Constructing a complete transformation with reusable transformation building blocks should be considerably less work than writing it from scratch. A transformation component should be a self-contained unit and composition should be as easy as possible without the need for much glue code.*

Neither ATL or MTF have an explicit notion of (external) transformation composition; they only support low-level internal composition of individual transformation

rules (see Section 2.2). Some other transformation languages have limited high-level composition support (QVT, VMT [15]) but cross-technology composition is generally hard to accomplish.

**External specification** *Each transformation should clearly specify what it requires from the environment and what it provides.*

An ideal external specification should only provide detailed information about the expected input and provided output models of a transformation (its environment). All other information such as metamodel choice, initial rule, invocation method, etc. is not relevant at this level and should be hidden in the specification. The external specification is only weakly defined in current transformation technologies (at many points implementation knowledge is required) and differs profoundly among the different technologies (see Section 2.1).

## 4 Unified Transformation Representation

In this section we describe a common, model-based, representation for transformations (Unified Transformation Representation – UTR) to establish a common ground among different transformation technologies. The UTR separates implementation, specification and execution of transformations and contains concepts to type and compose transformations (see Section 4.1). We also show how the UTR metamodel gives rise to a number of transformation roles in Section 4.2 and discuss a practical usage scenario that involves these roles in Section 4.3.

### 4.1 UTR Metamodel

Figure 2 shows an extract of the UTR metamodel. We clarify this metamodel in the following paragraphs.

**Basic Transformation Concepts** We make a distinction between the *specification* of a transformation, which encapsulates a concrete *implementation*, and the *execution* or instance of a transformation, which is subject to composition.

Each *TFSpecification* is characterized by an implementation and an external specification. An *AtomicTFSpecification* defines the transformation directly in terms of a specific transformation technology. Alternatively, a *CompositeTFSpecification* is itself expressed as a chain of subtransformations (inherits from *TFChain*). Implementation details, whether atomic or composite, are hidden by subclassing (*ATLImpl*, *MTFImpl* and *JavaImpl*). Notice that it is not our intention to allow multiple implementations for one *TFSpecification* but rather to provide a clear specification that hides intricate technology and implementation details.

A *TFSpecification* is represented in terms of input and output models, denoted by *TFFormalParameters* and their respective model types, denoted by *ModelingPlatforms*. We explain the concept of *ModelingPlatform* in greater detail later in this section, for now it suffices to see it as a constrained metamodel. The combination of *TFFormalParameters* and *ModelingPlatforms* specify exactly what a *TFSpecification* requires from

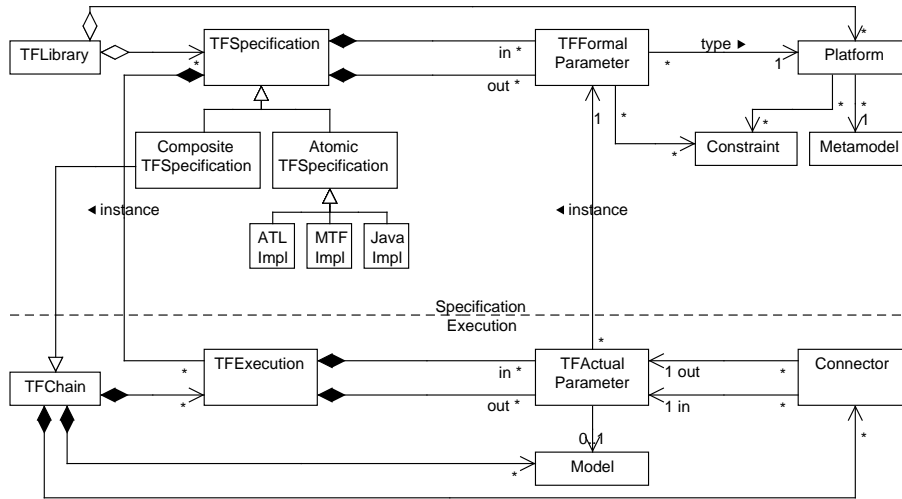


Fig. 2. Extract of the UTR metamodel.

its inputs and provides on its outputs and hence defines the context in which a transformation can be meaningfully executed. *ModelingPlatforms* and *TFSpecifications* can be grouped and organized in *TFLibraries*.

The lower part of Figure 2 represents the execution level. *TFExecution* and *TFActualParameter* are the runtime counterparts (or instances) of *TFSpecification* and *TFFormalParameter*. *TFActualParameter* is a container for a concrete *Model*, so only at this level we can execute a transformation. We introduce the *Connector* element to interconnect two *TFExecutions* through their *TFActualParameters*. The *Connector* is modeled as a class so that it can provide additional behavior such as verifying models, pause execution, etc. A connection is valid only if the *TFActualParameters* have the same type (equal *ModelingPlatforms*). Finally, the *TFChain* class represents a transformation chains that is composed of *TFExecutions*, *Connectors* and *Models*.

**Model Types** As discussed in Section 2.2, a metamodel is not always sufficient to accurately specify inputs and outputs of a *TFSpecification*. We need to be able to tune existing metamodels such as UML to our needs (metamodel variation and delimitation) and to enforce local transformation requirements (model structure).

In classical programming languages, similar problems can be encountered when specifying the signature of an operation. Some programming languages offer as a solution preconditions and invariants (Design by Contract [16]) that limit the range of allowed parameter values. We propose a similar approach to solve the limitations of typing by metamodel. We allow additional constraints on each transformation input so that we can exactly describe what is expected from the input model, beyond the structure imposed by the metamodel.

The specification of input and output models is accomplished in two stages (see Figure 2): (1) through the transformation-specific *TFFormalParameters* and (2) through cross-transformation reusable *ModelingPlatforms*. Both can impose *Constraints* on the models. We must stress that we do not refer to execution platforms such as J2EE or .NET in any way. A *ModelingPlatform* is composed of a *Metamodel* along with a number of *Constraints* that describe our expectations on the structure of the model beyond those captured in the metamodel. Such constraints are typically expressed using OCL [17]. In this way *ModelingPlatforms* allow for a more controlled reuse of existing meta-models. A *TFFormalParameter* is typed by a *ModelingPlatform* and can impose additional, transformation-local, constraints on the global *ModelingPlatform*.

Although *Constraints* can be attached to both *TFFormalParameters* and *ModelingPlatforms*, a number of good practices can be formulated. Constraints that are common to many transformations (via their *TFFormalParameters*) should be factored out to a shared *ModelingPlatform*. The mere fact that transformations have overlapping constraints is a sign to consider the introduction of an additional *ModelingPlatform*. Furthermore, *ModelingPlatforms* should only constrain constructs at the metamodel level (e.g. UML without inheritance). *TFFormalParameters* on the other hand constrain constructs at the model level (e.g. a class named ‘Proxy’ needs to be present). In summary, a *ModelingPlatform* is a globally or company-wide reusable entity while *TFFormalParameters* specify additional and local transformation-specific requirements.

## 4.2 Transformation Roles

In the previous subsection we have made a distinction between the following levels of model transformations: transformation specification, implementation and execution. We associate a different set of skills with each of these levels. Therefore we identify three roles in the development of transformations. The roles are loosely based on the roles of Knowledge Builder, Facilitator and User as proposed in [18].

**Transformation Developer** A person in this role is responsible for implementing transformations using the most appropriate technique (ATL, JAVA, etc.). In many cases this role will be anonymous, for example when reusing third-party transformations.

**Transformation Specifier** The Transformation Specifier is responsible for the external specification of a transformation. Depending on the context, this role performs a different task, hence we split it up in two subroles:

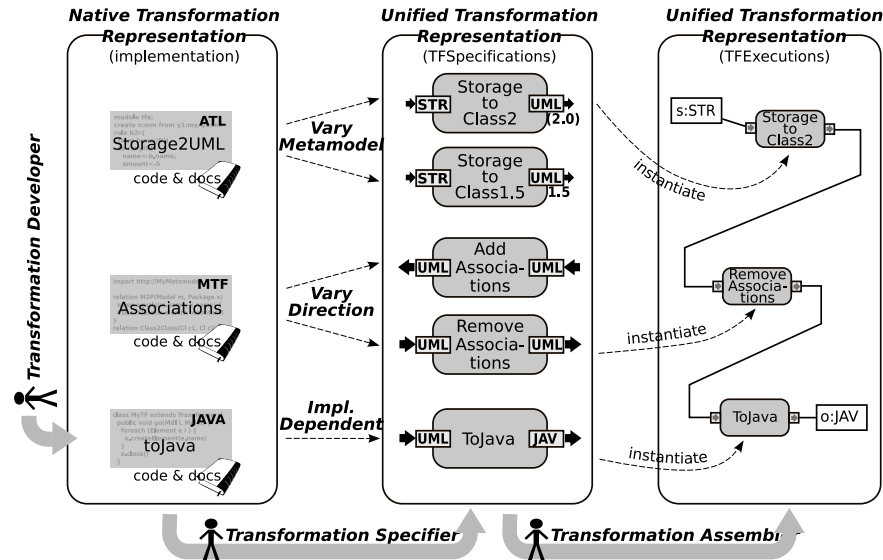
The **designer** subrole is used in the classical specification-before-implementation sequence. The designer gathers the requirements and defines the external specification of the required transformations. This information is then passed on to a Developer, who implements the given *TFSpecification*.

Because many transformation implementations are readily available we also introduce the more pragmatic role of the **harvester**. In this subrole the Specifier searches for available and appropriate transformation implementations that could be of use in a project. The harvester thoroughly studies and tests the implementations (a reverse engineering activity) so that appropriate *TFSpecifications* can be created. Basically this involves creating an explicit manifestation of the implicit assumptions that are present in the implementation.

**Transformation Assembler** This role selects appropriate *TFSpecifications* and composes *TFExecutions* into a transformation chain that realizes an overall transformation goal. The Assembler is shielded from low level implementation details because only *TFSpecifications* (provided by the Specifier) must be looked at.

### 4.3 Usage Scenario

To get a better view on how UTR concepts can be used and where the different roles come in, we present a typical usage scenario (see Figure 3). We describe the creation of a transformation chain (TFChain) that takes a *Storage* model as input and produces a corresponding JAVA model. The input model specifies a storage application such as a library or warehouse in a domain specific modeling language.



**Fig. 3.** A typical usage scenario. Part of the notation is derived from UML Activity diagrams.

The process of building the transformation chain consists of three steps. **First**, the Transformation Specifier/Harvester searches for candidate transformation implementations that can contribute to the chain. We have selected three relevant transformation implementations (left column of Figure 3): one converts Storage to UML models (ATL), one changes Associations within the UML model (MTF) and one transforms UML to equivalent JAVA models (JAVA). The only information that is available at this point is the source code of the transformation and textual documentation. As explained in Section 2, we need to fix many variables before we can execute any of the transformations.

In a **second step** we translate the transformation implementations into UTR Transformation Specifications. The Transformation Specifier is again responsible for this

step. Due to the variability points of the implementations (e.g. direction, metamodel), there are often several alternative but valid Specifications possible. For example, in the middle column of Figure 3, we have defined a separate Specification for each direction of the omnidirectional MTF transformation *Associations: AddAssociations* and *RemoveAssociations*. In the case of ATL (*Storage2UML*) we have varied the metamodels to target UML2.0 or UML1.5. The latter two are materialized as *ModelingPlatforms*.

The **final step** is to create the actual transformation chain from the building blocks provided by the previous steps. We are completely shielded from transformation implementation details at this stage. The Transformation Assembler instantiates the appropriate *TFSpecifications*, which yield *TFExecutions*, and composes them using *Connectors* (see right of Figure 3). Mind that Connectors can only connect *TFActualParameters* of the same type: their constraints must be compatible and their *ModelingPlatforms* must be the same. Finally we provide the input model *s* and a container for the result *o*.

The discussed scenario presents a pragmatic approach that leverages existing transformation implementations and represents them in UTR. Alternatively, we can start by defining *TFSpecifications* (middle column) based on a set of given requirements. These *TFSpecifications* are then implemented (left column), reused from previous projects or bought. Finally a transformation chain is created (right column). In practice a combination of both the pragmatic and the latter approaches can be appropriate.

## 5 Implementation

We have developed a tool that implements the Unified Transformation Representation and supports the usage scenario described in the previous section. This tool is called Unified Transformation Infrastructure (UniTI) and is built as a plugin for the Eclipse [19] platform. We have used the Eclipse Modeling Framework (EMF) [10] to represent the UTR metamodel. A dedicated model editor assists both Transformation Specifier and Assembler with the creation of transformation chains.

Figure 4 gives an impression of UniTI. The transformation chain that is shown is the same as in the usage scenario of Section 4.3. The left part of the figure shows a library of *TFSpecifications*. A number of so-called *wizards* assist the Transformation Specifier by asking for the necessary implementation details of each technology. *TFSpecifications* are then automatically generated. Note that the Transformation Specifier can now alter the generated *TFSpecifications* to his likings without destroying the coupling with the underlying implementation. Typically, *ModelingPlatforms* are refined at this point – e.g. the *BasicUMLClasses ModelingPlatform* in the figure.

If we shift focus to the right of Figure 4, we see the actual transformation chain model. Before modeling can start we need to import the necessary *TFLibraries* – only ‘MyTfLibrary’ in this case. We can now instantiate the necessary *TFExecutions* and connect them appropriately. The transformation chain can then be executed. Execution of each transformation is taken care of behind the scenes and hides technology-specific details. All intermediate models are automatically saved and are available for inspection during and after execution. UniTI supports parallel execution of transformations by allowing multiple *Connectors* on the same *TFActualParameter*. Conditional execution is not yet supported.

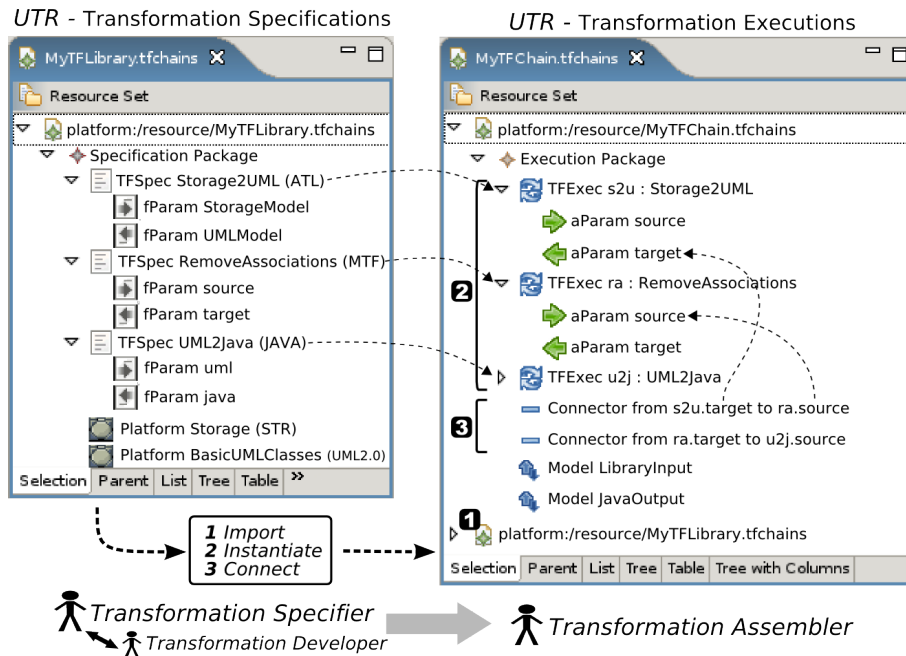


Fig. 4. An example worked out with UniTI.

We allow extensions of UniTI with new transformation technologies through two mechanisms. A *lightweight* extension mechanism is provided by the standard support for JAVA transformations. A simple JAVA interface that represents JAVA transformations can be reused to encapsulate any transformation technology that provides a JAVA API. A more *heavyweight* approach is by extending the metamodel by subclassing the Atomic Transformation Specification class. The main advantage of the heavyweight approach is better integration in UniTI; for example, we can provide a technology-specific wizard that simplifies the creation of Transformation Specifications.

## 6 Related Work

In our work we consider models and transformations as course-grained building blocks of an MDD approach. In the following paragraphs we discuss other approaches that apply a similar point of view.

In [2], transformation composition is seen as the composition of different tools that support a number of dedicated transformations or provide a generic transformation specification facility. An Eclipse plugin, the Model Control Center (MCC) allows the creation of transformation *networks* by using a simple scripting language that allows sequential, parallel and conditional execution. Our approach is similar to [2] by making a clear distinction between the specification and execution of transformations and al-

lowing different transformation technologies to work together. Our approach differs in the way we specify transformations. They reuse `JAVA` interfaces to encapsulate model types while we have the dedicated concept of Transformation Parameters and Platforms. We make a more detailed comparison of UniTI and MCC in the next section.

Another approach similar to ours is described in [20]. They use a classical component system as a Transformation Composition Framework (TCF). Transformations and models are encapsulated in components and component interfaces act as model types, execution entry points and context information nodes. They provide a basic transformation language for defining simple transformations; composite transformations are facilitated by the existing component framework. As in [2], transformations/models are typed by regular interfaces which limits their preciseness. They are also limited to their own transformation language, although this limitation seems easy to circumvent. Most of the limitations of this approach are a direct cause of reusing an existing component framework. UniTI avoids many of these limitations by offering more dedicated constructs. TCF will be compared to our approach in more detail in the next section.

In [21] a Domain Specific Language (DSL) is described to compose `JAVA` metamodel-specific transformation classes and automatically choose the right transformation instance depending on the preceding transformations. Our approach is similar to this one in that we also use a kind of DSL, which is defined by the UTR metamodel.

OMG's QVT [7] supports chaining of internal (QVT native) and external (implemented in another language) transformations at the transformation language level. It is, however, up to the tool implementer to provide the mechanisms to call out to external transformations so the composition of subtransformations is limited in practice. QVT supports precise model types by metamodel and compliance kind (strict, effective). This approach is comparable to, but more restricted than, our notion of Platform.

A number of approaches make use of classical build tools such as ANT [22] to manage transformation compositions. Examples are AM3/ATL [12] and OMELET [23]; the latter tries to extend build tools with the notion of metamodel as data type in order to fit them better to transformations.

Finally the MDDI ModelBus [24] provides a middleware that enables the execution of all kinds of model services provided by different tools in a technology-neutral fashion. Transformation services offered through the ModelBus could be integrated in UniTI through the provided extension mechanisms.

In summary, we proposed a model-based solution for transformation reuse and composition while most other approaches reuse existing technology to facilitate transformation chaining. Hence, the main advantage of our approach is that the abstractions that we use map very well to the transformation domain, while this is not always the case for the other approaches. A more detailed comparison of UniTI, MCC, TCF and ANT is made in the next section.

## 7 Evaluation

In this section we evaluate UniTI by comparison with other approaches that offer support for composition and reuse of transformations. We also look at how regular transformation languages support modularization and composition of transformations.

	<b>UniTI</b>	<b>ANT</b>	<b>TCF</b>	<b>MCC</b>
Tf Module	TF Spec. (wizard)	N/A	Component	Eclipse plugin
Tf exchange/reuse	import library	N/A	copy component	install plugin
Model Typing	Platform	N/A	JAVA Interface	JAVA Interface
Tf Composition	model-based	script	component composition	script
Cross-technology	built-in	manual	manual?	manual
Glue code	7 model elements	~60 lines	~20 lines	~6 lines

**Table 1.** Qualitative comparison of UniTI, ANT, TFC and MCC

Table 1 shows a comparison of selected characteristics of UniTI, ANT, TCF and MCC. Although ANT cannot be considered as a dedicated transformation composition tool we have included it because it is used for that purpose in practice. For each of the characteristics we have indicated the corresponding element/value in each of the approaches. For example, the transformation module of UniTI is the TFSpecification while for TCF it is a regular component and for ANT there is no such concept.

In order to get a measure of the effort required for defining a transformation chain in each of these tools we have expressed our running example (see Subsection 4.3) in each of them. The amount of necessary glue code was then recorded as lines of code. Because UniTI does not offer a textual syntax we have recorded the amount of model elements that needed to be created instead. We looked at the transformation chain from the perspective of a Transformation Assembler. Therefore we never counted the specification of a transformation in the glue code if it was possible to leave it out.

From this comparison we can conclude that ANT offers the least support for the Transformation Assembler. UniTI, TCF and MCC offer substantial advantages in the area of model typing, chaining and exchange. Notice that UniTI is the only approach that takes a model-based approach. It offers transformation concepts as first-class entities (e.g. Transformation Specification) while others reuse existing notions such as components and Eclipse plugins to represent transformations. Therefore, UniTI has very expressive model typing, easy exchange of transformations and concise glue code. Furthermore, UniTI has built-in support for different transformation languages and is thus the only approach that also supports the task of the Transformation Specifier by automatically generating Transformation Specifications.

<i>Transformation chain</i>	<i>Rule-based</i>	<i>Modularized rule-based</i>
<code>transf GET</code> <code>transf SET</code> <code>connect GET.out,SET.in</code>	<code>transf getset</code> <code>rule getset</code> <code>from Property</code> <code>to Operation 'get'</code> <code>to Operation 'set'</code>	<code>transf getset</code> <code>rule get</code> <code>from Property</code> <code>to Operation 'get'</code> <code>rule set</code> <code>from Property</code> <code>to Operation 'set'</code>

**Table 2.** Modularization alternatives for transformations.

In this paper we have focussed on composition of complete transformations as building blocks. Since the modularization mechanisms offered by transformation lan-

guages are usually situated at a lower level, e.g. transformation rules, it is hard to make a meaningful comparison. Instead, we believe that both techniques have to be used in conjunction. In some cases it might be more opportune to modularize a transformation at the implementation level, while in others, composition of larger building blocks might be a good solution. In Table 2, we show alternative possibilities to modularize a transformation that adds accessor and mutator operations (i.e. getters and setters) to a Class, expressed in pseudo-code. The leftmost definition makes use of transformation chaining while the other definitions are expressed in a rule-based transformation language. The middle definition shows a naive implementation and the rightmost definition decomposes the transformation into two separate rules. For the transformation chaining approach, one line of code corresponds with one model element in UniTI.

At this point it is not clear which (de)composition strategy is best for different situations. We believe that both approaches can live in harmony, but more experience/research is required in order to define a set of guidelines that guide the developer in choosing the right strategy for each situation. A more thorough study is out of the scope of this paper. For a more detailed evaluation of the composition possibilities offered by rule-based transformation languages we refer to [25].

## 8 Conclusions and Future Work

A possible approach to implement complex transformations is by composing many simple subtransformations. We showed that this is not so easy to accomplish with current transformation technologies, certainly not when mixing different technologies. The main problem is the weak separation of specification and implementation of transformations, which requires a deep implementation knowledge both when reusing and composing transformations. Therefore we have proposed a model-based approach that offers a unified view on transformations with a clear distinction between implementation and specification. Transformation chains can thus be created without bothering with the technological or implementation details of each subtransformation.

The core concepts of our approach are defined in the Unified Transformation Representation (UTR) metamodel, which fulfills the following characteristics for transformations: the *black-box principle*, subject to *composition* and clear *external specification*. In order to realize these we made a clear distinction between implementation, specification and execution of transformations. Our approach gave rise to different usage roles: Transformation Developer, Specifier and Assembler. Each of these roles has a restricted view on transformations and is assigned with a clear set of responsibilities. The Developer implements a transformation in the most appropriate technology, the Assembler composes transformations without having to know anything about the underlying implementation and the Specifier mediates between the former two by providing Transformation Specifications. UniTI implements the UTR metamodel and assists both Transformation Specifier and Assembler to create transformation chains in a technology-transparent fashion.

In future work we will investigate how to offer cross-transformation services such as end-to-end traceability. We will explore how this traceability information can be used throughout the transformation chain to improve subsequent transformations.

## References

1. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15** (1972) 1053–1058
2. Kleppe, A.: Mcc: A model transformation environment. In: ECMDA-FA. (2006) 173–187
3. Jouault, F., Kurtev, I.: Transforming models with atl. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (2005)
4. IBM Alphaworks: Model transformation framework. Misc (2004) <http://www.alphaworks.ibm.com/tech/mtf>.
5. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA Visual Automated Transformations for Formal Verification and Validation of UML Models. In: Proceedings of the 17th IEEE international conference on Automated software engineering. (2002)
6. Ho, W.M., Jezequel, J.M., Pennanc'h, F., Plouzeau, N.: A toolkit for weaving aspect-oriented uml designs. In: Proceedings of the 1st Conference on Aspect-Oriented Software Development, ACM Press (2002,) 99–105
7. Object Management Group: Qvt-merge group submission for mof 2.0 query/view/transformation. Misc (2005)
8. Object Management Group: Uml 2.0 superstructure fit convenience document. Misc (2004)
9. Object Management Group: Meta object facility (mof) 2.0 core specification. Misc (2004)
10. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)
11. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the context of MDA. (2003)
12. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: The AMMA platform support for modeling in the large and modelling in the small. Technical Report 04.09, LINA (2004)
13. Barbero, M., Fabro, M.D.D., Bézivin, J.: Traceability and provenance issues in global model management. In: 3rd ECMDA-Traceability Workshop. (2007)
14. Szyperki, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Professional (1997)
15. Sendall, S., Perrouin, G., Guelfi, N., Biberstein, O.: Supporting model-to-model transformations: The vmt approach. Technical report (2003)
16. Meyer, B.: Applying "design by contract". *Computer* **25** (1992) 40–51
17. Object Management Group: Uml 2.0 ocl final adopted specification. Misc (2003)
18. Gavras, A., Belaunde, M., Almeida, L.F.: Towards an mda-based development methodology. In: EWSA. (2004) 230–240
19. Beaton, W., d. Rivieres, J.: Eclipse platform technical overview. Technical report, The Eclipse Foundation (2006)
20. Marvie, R.: A transformation composition framework for model driven engineering. Technical Report LIFL-2004-10, LIFL (2004)
21. Wagelaar, D.: Blackbox composition of model transformations using domain-specific modelling languages. In: ECMDA-FA 2006 workshop on Composition of Model Transformations. (2006)
22. Moodie, M.: Pro Apache Ant (Pro). Apress, Berkely, CA, USA (2005)
23. Willink, E.D.: Omelet:exploiting meta-models as type systems. In: 2nd European Workshop on MDA with an emphasis on Methodologies and Transformations. (2004)
24. Blanc, X., Gervais, M.P., Sriplakich, P.: Model bus: Towards the interoperability of modelling tools. In: MDAFA. (2004) 17–32
25. Kurtev, I., van den Berg, K., Jouault, F.: Evaluation of rule-based modularization in model transformation languages illustrated with atl. In: Proceedings of the 2006 ACM symposium on Applied computing. (2006) 1202–1209