Realizing Model Simplifications with QVT Operational Mappings

Alexander Kraas

Poppenreuther Str. 45, D-90419 Nürnberg, Germany
alexander.kraas@gmx.de

Abstract. After parsing the input of a textual modeling language, further processing steps may be required before the result can be mapped to corresponding elements in a model. For instance, such a processing step can be the simplification of syntactic constructs. An approach for model simplification resting on transformation patterns is presented in this paper. The presented approach rests on the refinement of a derived base transformation with the superimposition of mapping operations. The transformations are specified with the Operational Mappings part of the Query/View/Transformations (QVT) specification.

Keywords: QVT-O, Transformation, Patterns, Simplification

1 Introduction

In general, modeling languages can be classified into textual as well as graphical modeling languages. However, also hybrid approaches using both kinds of modeling exist. For instance, the Specification and Description Language (SDL) [9] can be considered as such a language. Further processing steps (e.g. simplifications) may be required after the textual input of a hybrid modeling language is parsed and represented in terms of a Concrete Syntax Tree (CST). According to the taxonomy given in [1], model simplification is an approach where particular syntactic constructs are transformed into more primitive constructs. Usually, dedicated frameworks for language transformations, such as Stratego/XT [7], can be utilized for this purpose. However, an access to an already existing model may be required for the simplification of the textual input of a hybrid language. If a CST is defined in terms of a metamodel, the Query/View/Transformations (QVT) [8] specification supports such an transformation scenario.

Since the general realization of transformation patterns by using the Relational Language of QVT is already discussed in [3], in this paper the implementation of patterns for model simplification by using the Operational Mappings of QVT is discussed. The QVT features transformation extension and superimposition of mapping operations play a key role for the presented approach. Even if these features are defined in the QVT specification [8], only less information concerning their combined application can be found in the literature. Hence, the successful application of the approach for the textual SDL editor of the SUMoVal framework [10] is used in this paper to illustrate the implementation of

simplification patterns. The transformations within the SU-MoVal framework are executed by the QVT operational component (QVTo) [12] of Eclipse.

The rest of this paper is structured as follows. The proposed approach and transformation patterns for model simplification are introduced in section 2. Related work is discussed in section 3 and a conclusion is given in section 4.

2 Transformation Patterns for Model Simplification

An approach to implement patterns for model simplification with QVT Operational Mappings (QVT-O) is presented in this chapter. After the general approach is discussed, a transformation example is introduced in section 2.2, which is used to explain the approach more descriptive in subsequent sections. Finally, the realization of exemplary patterns for model simplification is discussed in section 2.4.

2.1 General Approach

The presented approach to simplify models by using QVT Operational Mappings (QVT-O) rests upon a common endogenous base transformation (T_{CB}) that is extended by another transformation (T_{SP}) implementing a particular simplification pattern.

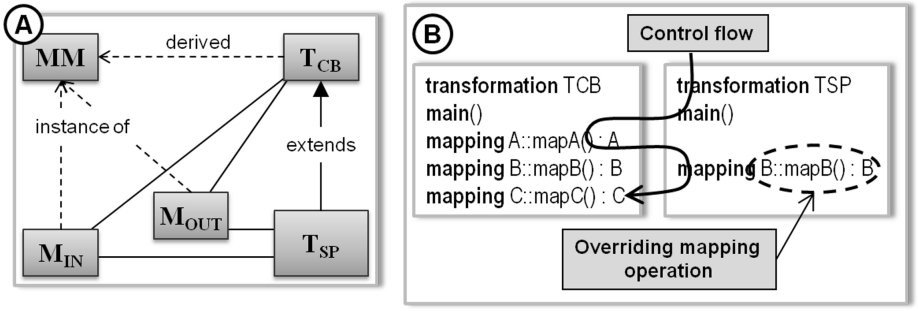

Fig. 1. Conceptual overview

Derivation of a common base transformation: As shown in part A of Fig. 2, the required transformation T_{CB} is derived from a metamodel (MM) and the input model (M_{IN}) and the output model (M_{OUT}) are instances of MM. The purpose of T_{CB} is to produce only a one-to-one copy of model M_{IN} and therefore, a particular mapping operation for each metaclass is contained in T_{CB}. Even if a higher-order transformation could be applied to derive T_{CB} from a metamodel MM, transformation T_{CB} is generated with a Model-to-Text (M2T) approach so that its source code can be partially reused to specify transformation T_{SP}.

In order to achieve that transformation T_{CB} makes a copy of M_{IN}, all elements of M_{IN} are mapped to equal elements in M_{OUT}. Hence, T_{CB} implements an entire rewrite system that consists of contextual mapping operations that invoke other mapping operations in a nested manner to map the properties of an input element.

Realizing a simplification pattern: When a model simplification pattern is implemented by transformation T_{SP}, the control flow of T_{CB} has to be redirected at points where the relevant elements to be simplified occur. Usually, such points are mapping operations that are defined in the context of the relevant metaclasses. After the mapping operations in T_{CB} are identified, they have to be overridden by corresponding mapping operations (see part B of Fig. 2) in T_{SP}, which implement the desired simplification pattern.

2.2 Transformation Example and Required Metaclasses.

In order to make the approach discussed in the following sections more descriptive, a transformation example resting on a metamodel for representing parse-trees of the concrete syntax (`CS`) of the Specification and Description Language (SDL) [9] is used. This example is taken from the SDL-UML Modeling and Validation framework (SU-MoVal) [10].

Since the concrete syntax of SDL makes use of so-called short hand notations that have to be transformed to equivalent language constructs, this is considered to be a good case study for model simplification. In particular, the transformation example consists of the transformation of a mathematical or logical in-line operator to a corresponding operation application. For instance, the concrete syntax expression `1 + 1` is transformed to `"+"(1, 1)`.

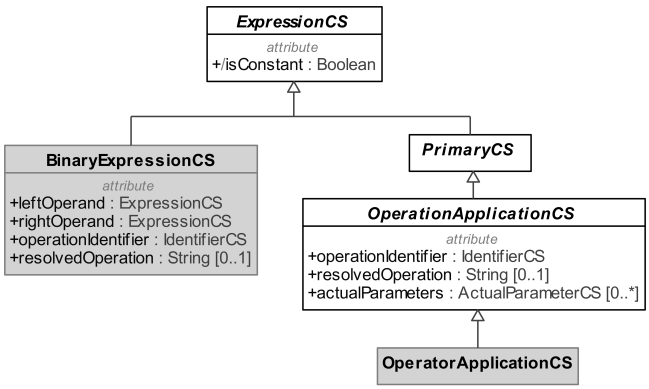

Fig. 2. Binary expression metaclass

As shown in Fig. 2, the `BinaryExpressionsCS` metaclass is a sub-type of `ExpressionCS` and it is used to represent mathematical or logical inline opera-

tors (e.g. + operator) within SDL expressions. After application of the transformations patterns discussed in the subsequent sections, an instance of `BinaryExpressionCS` is transformed to an `OperatorApplicationCS`.

2.3 Derivation of a Common Base Transformation

The derivation of a transformation from a metamodel by using a Model-to-Text (M2T) transformation is out of scope of this paper. Hence, the derivation approach is only discussed on a high-level of abstraction. For instance, the M2T tool Acceleo [11] is used for the derivation task during the implementation of SU-MoVal [10].

Transformation Main Part. As shown in the code example below, the required `main` operation of T_{CB} is empty, because the concrete behavior is specified by a transformation that extends T_{CB}. Furthermore, the input and output parameters of T_{CB} have the same type because T_{CB} is an endogenous transformation.

```
modeltype CS uses ConcreteSyntax ("http://...");
transformation TCB(in inp:CS, out outp:CS)
main() { // Nothing to do here !}
```

Mapping Operations for Metaclasses. For each metaclass of the example metamodel `CS`, a corresponding mapping operation for T_{CB} is derived. Depending on the kind of a metaclass (abstract or non-abstract), two different kind of mapping operations are generated. In addition, the context (`self`) and the `result` type of a mapping operation correspond to the currently processed metaclass.

For each abstract metaclasses, a disjunctive mapping operation is introduced consisting of an ordered list of mapping operations for all subclasses of that metaclass (e.g. `ExpressionCS`). An important fact is that attributes, if any, of an abstract metaclass cannot taken into account by a disjunctive mapping operation. Instead, these attributes are processed by each mapping operation listed as disjunctive alternative.

```
mapping CS::ExpressionCS::mapExpressionCS() : CS::ExpressionCS
disjuncts
  CS::EqualityExpressionCS::mapEqualityExpressionCS,
  CS::PrimaryCS::mapPrimaryCS,
  CS::TypeCoercionCS::mapTypeCoercionCS,
  CS::MonadicExpressionCS::mapMonadicExpressionCS,
  CS::CreateExpressionCS::mapCreateExpressionCS,
  CS::RangeCheckExpressionCS::mapRangeCheckExpressionCS,
  CS::BinaryExpressionCS::mapBinaryExpressionCS {}
```

In contrast to abstract metaclasses, all owned and inherited attributes of a non-abstract metaclass (e.g. `BinaryExpressionCS`) are processed for the derivation of the body of a corresponding mapping operation.

```
mapping CS::BinaryExpressionCS::mapBinaryExpressionCS()
 : CS::BinaryExpressionCS {
   result.resolvedOperation := self.resolvedOperation;
   result.operationIdentifier := self.operationIdentifier.map
      mapIdentifierCS();
   result.leftOperand  := self.leftOperand.map mapExpressionCS();
   result.rightOperand := self.rightOperand.map mapExpressionCS();
   result.resolvedType := self.resolvedType;
}
```

Processing of Metaclass Attributes. In general, the mapping operation for a non-abstract metaclass assigns each owned or inherited attribute of an input object (`self`) to a corresponding attribute of the result object (`result`). Since QVT-O distinguish between assignments to mono- or multi-valued properties or variables, this has also to be considered during the processing of attributes. Hence, a simple assignment (`':='` operator) is used for the mapping of an attribute with an upper multiplicity of 1. In contrast, attributes with an upper multiplicity of >1 are assigned with a composite assignment (`'+='` operator).

```
result.myProperty := self.myProperty.map mapMySubclassA();
result.myProperty += self.myProperty->map mapMySubclassA();
```

The attribute type is another property that is taken into account during the derivation process. Usually, a type dependent mapping operation is invoked, before the value is assigned to the output object. This is not the case for attributes with a primitive (e.g. `String`) or an enumerated type for which the input value is just copied to the corresponding attribute of the output object.

```
result.myProperty := self.myProperty;
result.myProperty += self.myProperty;
```

Design Principles and Used Features of QVT-O. Apart from the discussed extension mechanism also the access mechanism of QVT-O could be considered to access mapping operations of T_{CB}. However, this mechanism does not support transformation inheritance so that the transformation control flow cannot be redirected through superimposition of mapping operations of T_{SP}.

Furthermore, QVT-O supports the inheritance and the merge of mapping operations. These features where not taken into account for the presented approach, because they induce a combined execution of all involved mapping operations. If these features would be used, the mapping rules for a particular metaclass had to be specified in different mapping operations. Instead, the discussed approach recommends to process all inherited and owned attributes of a metaclass

within one mapping operation of T_{CB}. The advantage for the implementation of transformation patterns is a better maintainability and a lower complexity, because required mapping rules can be specified within one overriding mapping operation of T_{SP}.

Since the inheritance and the merge of mapping operations are not used, also the QVT-O feature of abstract mapping operations is not required in order to process abstract metaclasses of MM. Instead, disjunctive mapping operations are used for this purpose, because they make a fine grained manipulation of the transformation's control flow possible.

2.4 Exemplary Transformation Patterns for Model Simplification

Resting on the already introduced approach (see Sec. 2.1), exemplary transformation patterns for model simplification are discussed in more detail in this section. For this purpose, the transformation example of section 2.2 is used in the following paragraphs.

Top-level Simplification. A model can be represented as a tree structure of nested model elements, which are instances of different metaclasses of a metamodel. The most simple use case for model simplification is the transformation of top-level elements only. If this pattern is applied to the transformation example (see Sec. 2.2), only the top-level instances of `BinaryExpressionCS` will be transformed to a corresponding `OperatorApplicationCS` and nested `BinaryExpressionsCS` will be preserved (see example output O_1 shown in Fig. 3).

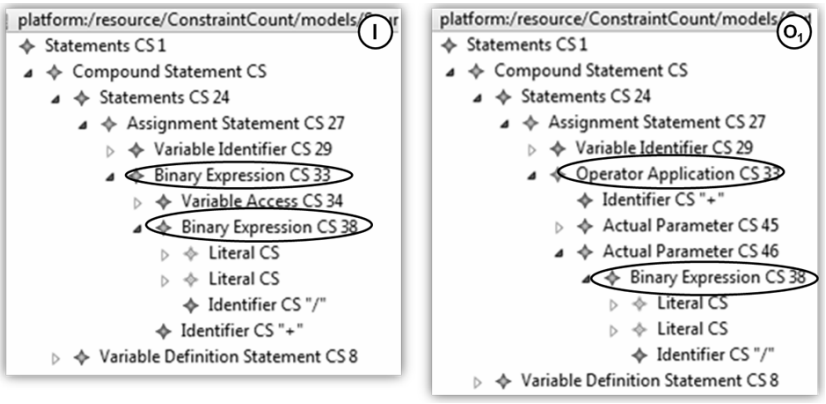

Fig. 3. Example input model and output model 1

In order to realize this pattern for the given transformation example with QVT, the transformation T_{CB} is extended by transformation `InfixOperationToOperator`. Within the `main()` method of this transformation, the root element

of the model is selected and an appropriate mapping operation of the T_{CB} transformation is invoked. According to the presented approach, the control flow of the transformation remains in T_{CB}, as long as one of its mapping operations is overridden by another operation specified in the `InfixOperationToOperator` transformation.

```
transformation InfixOperationToOperator
  (in input : CS, out output : CS) extends TCB;
main() {
  input.rootObjects()[CS::StatementCS]->map mapStatmentCS(); }
```

In order to transform each occurrence of `BinaryExpressionsCS`, the control flow within T_{CB} is redirected at points where associated mapping operations are invoked. Therefore, the operation `mapExpressionCS()` is overridden as follows:

```
mapping CS::ExpressionCS::mapExpressionCS() : CS::ExpressionCS
disjuncts
  CS::EqualityExpressionCS::mapEqualityExpressionCS,
  CS::PrimaryCS::mapPrimaryCS,
  CS::TypeCoercionCS::mapTypeCoercionCS,
  CS::MonadicExpressionCS::toOperatorApplication,
  CS::CreateExpressionCS::mapCreateExpressionCS,
  CS::RangeCheckExpressionCS::mapRangeCheckExpressionCS,
  CS::BinaryExpressionCS::toOperatorApplication {}
```

The last line (bold printed) of `mapExpressionCS()` is modified in comparison to the overridden operation in T_{CB}. That is because the control flow shall be redirected to the `toOperatorApplication()` operation that implements the mapping of a `BinaryExpressionCS` to an `OperatorApplicationCS`.

```
mappingCS::BinaryExpressionCS::toOperatorApplication()
  : CS::OperatorApplicationCS {
  result.resolvedType := self.resolvedType;
  result.resolvedOperation := self.resolvedOperation;
  result.operationIdentifier := self.operationIdentifier;
  result.actualParameters := OrderedSet {
    object ActualParameterCS { expression := self.leftOperand },
    object ActualParameterCS { expression := self.rightOperand }};
}
```

Since only top-level instances of `BinaryExpressionCS` shall be transformed by the `toOperatorApplication()` operation, all attributes of an input element are assigned directly to the created output element (without invoking any other mapping operation).

Recursive Simplification. The objective of the pattern for recursive simplification is to transform all model elements of a particular kind. This is realized with a recursive call of the mapping operation, if required. After applying a recursive simplification pattern for the given transformation example, the input model `I` (shown in Fig. 3) is transformed to the output model `O2` (see Fig. 4). As expected, all instances of `BinaryExpressionCS` in the output model `O2` are transformed to instances of `OperatorApplicationCS`.

In order to implement the recursive transformation pattern for the given transformation example, a slightly modified variant of the already discussed `InfixOperationToOperator` transformation is used. Hence, the `toOperatorApplication()` mapping operation is modified in a way so that also for nested elements dedicated mapping operations are invoked instead of copying them.

```
result.actualParameters :=  OrderedSet {
  object ActualParameterCS { expression :=
    self.leftOperand.map mapExpressionCS() },
  object ActualParameterCS { expression :=
    self.rightOperand.map mapExpressionCS() } }
 };
```

The additional map operation calls (bold printed) in the code snippet shown above realize the recursive call to the `toOperatorApplication()` mapping operation. When one of the `mapExpressionCS()` operations is invoked, appropriate mapping operations defined in T_{CB} are processed as long as an instance of `BinaryExpressionCS` shall be mapped. If this is the case, the control flow of the transformation is redirected to the `toOperatorApplication()` operation once again.

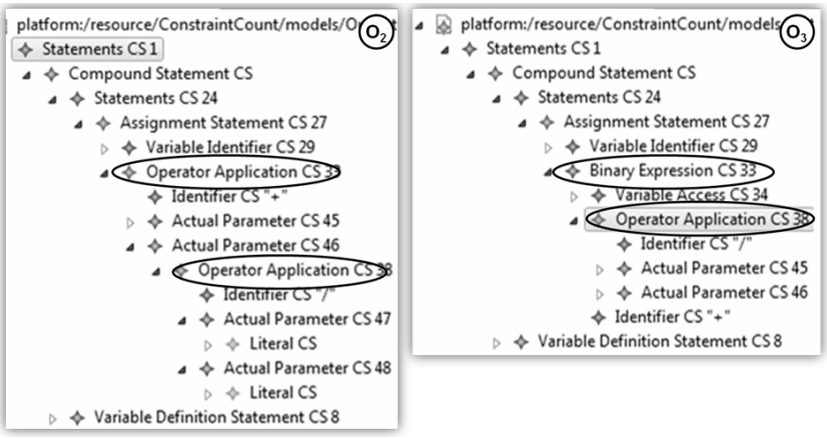

Fig. 4. Output models 2 and 3

Simplification of Leaf Elements. The transformation pattern for simplification of leaf elements can be considered as opposite of the already presented top-level simplification pattern. A simplification of leaf elements applied to the transformation example causes a mapping of leaf `BinaryExpressionCS` elements to corresponding `OperatorApplicationCS` elements, whereas top-level elements of that kind are only copied to the output model (see output model `O3` in Fig. 4).

A modified variant of the `InfixOperationToOperator` transformation can be used to implement the simplification of leaf elements. Therefore, the `mapBinaryExpressionCS()` operation, which just makes a copy of the input element, is added as an additional mapping alternative to the disjunctive mapping operation `mapExpressionCS()`. It is important to add `mapBinaryExpressionCS()` after `toOperationApplication()`, because the disjunctive alternatives are processed in sequential order.

```
mapping CS::ExpressionCS::mapExpressionCS() : CS::ExpressionCS
disjuncts
  ...
  CS::BinaryExpressionCS::toOperatorApplication,
  CS::BinaryExpressionCS::mapBinaryExpressionCS {}
```

A **when** clause is added to the `toOperatorApplication()` operation, because it shall only be invoked for input elements that have no further nested `BinaryExpressionCS` elements. If the condition of the when clause is not fulfilled, the mapping operation is not invoked.

```
mappingCS::BinaryExpressionCS::toOperatorApplication()
  : CS::OperatorApplicationCS
when { self.allSubobjectsOfType
        (CS::BinaryExpressionCS)->size() = 0 }
```

3 Related Work

Many works [2, 3, 6] concerning the refinement of endogenous transformations and the implementation of transformation patterns with the QVT Relational Language (QVT-R) exist. In addition, the usage of the AtlanMod Transformation Language (ATL) for this purpose is analysed in [4, 6]. Even if the mentioned works discuss possible approaches, the analysis of the ATL Transformation Zoo in [5] comes to the conclusion that transformation superimposition is rarely used in practice.

In contrast to before mentioned works, the approach presented in this paper rests on the QVT Operational Mappings (QVT-O) [8], which is an imperative transformation language and not a relational language as used by the mentioned works. Furthermore, instead of introducing only a copy pattern for QVT-O transformations, the presented work discuss the refinement of generated copy patterns towards essential patterns for model simplification.

4 Conclusion

The approach for model simplification by using the QVT Operational Mappings presented in this paper is not only restricted to the used metamodel of the textual notation of SU-MoVal [10]. That is because the rules for the derivation of a common base transformation, which is the starting point for all discussed simplification patterns, are also applicable to each other kind of metamodel.

In addition, the applicability of the approach is not limited to the patterns discussed in the section before, because further patterns for simplification can be realized in a similar manner. As a matter of principle, also other kind of endogenous model transformations could be implemented in the same manner, because the area of model simplification is only used as an exemplarily use-case in order to demonstrate the overall approach. A running transformation example and further information can be obtained from the SU-MoVal homepage [10].

References

1. Mens T., Van Gorp P.: A Taxonomy of Model Transformation. In: Electronic Notes in Theoretical Computer Science, vol. 152, pp. 125 – 142. Elsevier, Amsterdam (2006)
2. Goldschmidt T., Wachsmuth G.: Refinement transformation support for QVT Relational transformations. In: 3rd Workshop on Model Driven Software Engineering, MDSE 2008 (2008)
3. Iacob M-E., Steen M., Heerink L.: Reusable model transformation patterns. Enterprise Distributed Object Computing Conference Workshops, 2008 12th., pp. 1 – 12. IEEE Press, New York (2008)
4. Tisi M., et. al.: Refining Models with Rule-based Model Transformations. INRIA, Rennes, France (2011)
5. Kusel A., et. al.: Reality Check for Model Transformation Reuse: The ATL Transformation Zoo Case Study. In: 2nd Workshop on the Analysis of Model Transformations, AMT@Models'13, pp. 1 – 10. Miami (2013)
6. Wagelaar D., Van Der Straeten R., Deridder D.: Module superimposition: a composition technique for rule-based model transformation languages. In: Software & Systems Modeling, vol. 9, issue 3, pp. 285 – 309, Springer, Heidelberg (2010)
7. Bravenboer M., Kalleberg K. T., Vermaas R., Visser E.: Stratego/XT 0.17. A language and toolset for program transformation. In: Science of Computer Programming, vol. 72, issue 1, pp. 52 – 70, Elsevier (2008)
8. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1. OMG Document Number: formal/2011-01-01. http://www.omg.org/spec/QVT/1.1/PDF/
9. International Telecommunication Union: Recommendation Z.101 (12/2011), Specification and Description Language – Basic SDL-2010. https://www.itu.int/rec/TREC-Z.101/en
10. SDL-UML Modeling and Validation (SU-MoVal) framework, Open source software, http://www.su-moval.org/
11. Acceleo Model-to-Text transformation framework, Open source software, http://www.eclipse.org/acceleo/
12. QVT Operational component of Eclipse, Open source software, http://projects.eclipse.org/projects/modeling.mmt.qvt-oml