

Formalization and Rule-Based Transformation of EMF Ecore-Based Models

Bernhard Schätz

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
schaetz@in.tum.de

Abstract. With models becoming a common-place in software and systems development, the support of automatic transformations of those models is an important asset to increase the efficiency and improve the quality of the development process. However, the definition of transformations still is quite complex. Several approaches – from more imperative to more declarative styles – have been introduced to support the definition of such transformations. Here, we show how a *completely* declarative relational style based on the interpretation of a model as single structured term can be used to provide a transformation mechanism allowing a simple, precise, and modular specification of transformations for the EMF Ecore platform, using a Prolog rule-based mechanism.

1 Motivation

The construction of increasingly sophisticated software products has led to widening gap between the required and supplied productivity in software development. To overcome the complexity of realistic software systems and thus increase productivity, current approaches increasingly focus on a *model-based* development using appropriate description techniques (e.g., UML [1]). Here, the specification of a software product is described using *views* of the system under development (*horizontal*, e.g., structure, the behavior; *vertical*, e.g., component/sub-components). To combine those views, they are mapped onto a common model.

Besides view-based development, automated development steps – using rule-based mechanized transformations – are an important technique to improve the efficiency of the development process. Besides increasing efficiency, these transformations offer consistency ensuring modification of models, ranging from refactoring steps to improve the architecture of a system to the consistent integration of standard behavior.

In the following, a transformation approach is introduced, allowing a simple, precise, and modular specification of transformations. The approach uses a declarative relational style to provide a transformation mechanism. It is implemented on the Eclipse platform, using a term-based abstract representation of an EMF Ecore description and a Prolog rule-based interpretation.

1.1 Related Approaches

The introduced transformation framework is used to describe graph transformations, using a relational calculus focused on basic constructs to manipulate nodes (elements)

and edges (relations) of a conceptual model. In contrast to other graph-based approaches like MOFLON/TGG [2], VIATRA [3], FuJaBa [4], or GME [5] it is not based on graph-grammars or graphical, rule-based descriptions [6], but uses a textual description based on a relational, declarative calculus. As shown in Section 3, for the declaration of basic transformations the rule-based approach allows the definition of such transformations in a pre-model/post-model style of description similar to graph-patterns, both concerning complexity and readability of description. However, in contrast to those approaches, the approach introduced here uses only a single formalism to describe basic transformations as well as their compositions; thus, it avoids the problems of using different formalisms to describe transformation patterns, like object diagrams, OCL expressions, and state machines. Similarly, this relational rule-based approach makes the order of application and interaction of basic transformations explicit, avoiding the imprecision resulting from the underspecification of these dependencies.

Considering the textual form of description, the transformation framework is similar to the QVT approach [7] and its respective implementations like ATL [8], F-Logics based transformation [9], or Tefkat [10]. However, in contrast to those it comes with a precise and straightforward definition of models *and* transformations. This definition is directly reflected in the description of models and transformations: There is only a single homogenous formalism with two simple construction/deconstruction operators to describe the basic transformation rules and their composition; complex analysis or transformation steps can be easily modularized since there are no side-effects or incremental changes during the transformation. Furthermore, the rule-based relational approach allows to arbitrarily mix more declarative and imperative forms of specification, which are strictly separated in those other approaches. Thus, e.g., it allows to construct an initial description of a specification in a purely declarative fashion, and to successively rephrase it into a more imperative and efficient form.

Similar to other approaches like VIATRA [3] or GEMS [11], Prolog is used to implement a relational style of describing transformations. However, in those approaches the model is encoded as a *set of base clauses*. As a consequence, transformations are implemented via rules based on side-effects using an `assert/retract`, thus making a truly relational description of transformations impossible. This use of side-effects, however, complicates the definition of complex declarative transformation, especially when describing pattern-like rules potentially requiring backtracking. A similar encoding is also used in formalizing models instantiating meta-models as facts in a relational database schemes, with the schemes derived from the meta-model relations, as in [12]. In contrast, here the model is encoded as a single complex *term*, which forms an input and output argument of the Prolog clause formalizing the transformation, thus allowing to use true relational declarative style without side-effects, even for complex transformation constructed in a modular fashion from simpler transformations. Note that the approach is not restricted to the use of Prolog; also other formalisms like functional languages can be used.

Finally, the approach presented here addresses a different form of application: While most of the above approaches intend to support the transformation from one modeling domain to another (e.g., from class diagrams to DB schemata), here the goal is rather

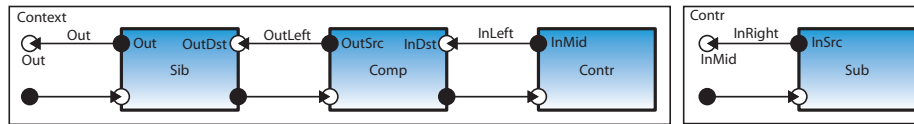


Fig. 1. Example: Hierarchical Component Model

to provide simple implementations for refactorings of models (e.g., combining a cluster of components).

1.2 Contribution

The approach presented in the following sections supports the transformation of EMF Ecore models using a completely *declarative relational* style in a *rule-based* fashion:

Relational: A transformation is described as a relation between the models before and after the transformation; models themselves are also described as relations.

Declarative: The description of the transformation relation characterizes the models before and after the transformation rather than providing an imperative sequence of transformation operations.

Rule-Based: The relations and their characterization of the models are described using a set of axioms or rules.

To support this style of description with its simple, precise, and modular specification of transformation relations on the problem- rather than the implementation-level, three essential contributions are supplied by the framework presented here:

1. The *term-structure representation* of models based on the structure of the meta-models they instantiate, as well its implementation in the Eclipse framework, providing the translation of models of EMF Ecore-based meta-models into their corresponding term-structure.
2. The *relational formalization of model transformations*, as well as its implementation using a Prolog rule-based interpretation, providing the execution of transformations within the Eclipse framework.
3. The *application of relations* as execution of – generally parameterized – transformations, as well as its implementation as an Eclipse plug-in, providing support for the definition and the execution of relational transformation rules.

These three contributions are described in the following sections. Section 2 provides a formalized notion of a model used in this relational approach as well as its representation in a declarative fashion using Prolog style. Section 3 describes the the basic principles of describing transformations as relations, using rules similar to graph grammars as a specific description style, and illustrates the modular composition of transformations. Section 4 illustrates the implemented tool support both for the definition of transformation and the transformation execution. Section 5 highlights some benefits and open issues.

2 Model Formalization and Structure

As mentioned in Section 1, the purpose of the approach presented here is the transformation of descriptions of systems under development to increase the efficiency and quality of the development process. According to the nomenclature of the OMG, the description of a system is called a (*system*) *model* in the following. Figure 1 shows such a model, as it is used in the AutoFOCUS [13] approach to describe the architectural structure of a system: the system Context, consisting of subcomponents Sib, Comp, and Contr, the latter with subcomponent Sub; the components have input and output ports like InDst and InMid, connected by channels like InLeft.

To construct formalized descriptions of a system under development, a ‘syntactic vocabulary’ also called *conceptual model* [13] is needed. This conceptual model¹ characterizes all possible system models built from the *modeling concepts and their relations* used to construct a description of a system; typically, class diagrams are used to describe them. Figure 2 shows the conceptual elements and their relations used to describe the architectural structure of a system in the AutoFOCUS approach. These concepts are reflected in the techniques used to model a system. In the following subsection, a *formalization of conceptual models and system models* based on relations is given as well as their representation in a declarative fashion using Prolog style.

2.1 Formalization

To define the notion of a *conceptual model* in the context of model transformations more formally, the interpretation along the lines of [13] is used. It provides a semantical interpretation for syntactical descriptions like in Figure 2. Basically, the conceptual model is constructed from a *conceptual universe*, containing the primitives used to describe a system: *concepts* characterizing unique entities used to describe a system, with examples in the AutoFOCUS conceptual model like *component*, *port*, or *channel* to define the components, ports and channels of a structural description of a system; *attributes* characterizing properties, like *name* or *direction* to define the name of a component and a channel, or the direction (input or output) of a port. Concepts and attributes form the conceptual universe, consisting of a collection of infinite sets of conceptual entities, and a collection of – finite or infinite – sets of attribute values. In case of the AutoFOCUS conceptual model, examples for sets of conceptual entities are $CompId = \{comp_1, comp_2, \dots\}$, and $PortId = \{port_1, port_2, \dots\}$; typical examples for set of attribute values are $CompName = \{\text{‘Sib’}, \text{‘Context’}, \dots\}$ or $PortDir = \{input, output\}$.

Based on the conceptual universe, the *conceptual domain* is defined, consisting of elements corresponding to objects used to model a system, like *Component*, *Port*, or *Channel* to define the components, ports and channels of a structural description of a system; and relations corresponding to dependencies between the elements, like *subComp*, *chanComp*, or *srcPort* to define the subcomponents of a component, the component containing a channel, or the source port of channel.

¹ In the context of technologies like the Meta Object Facility, the class diagram-like definition of a conceptual model is generally called *meta model*.

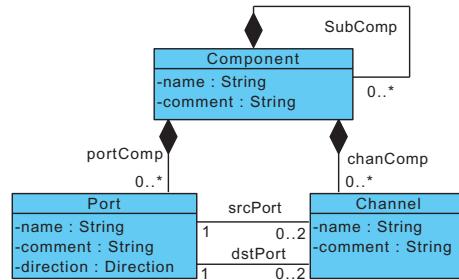


Fig. 2. Representation of the Conceptual Model of AutoFOCUS Component Diagrams

The conceptual domain consists of a collection of element relations between conceptual entities and attribute values, and a collection of (binary) association relations between conceptual entities. In case of the AutoFOCUS conceptual domain for structural descriptions as provided in Figure 2², examples for element relations are $Component = CompId \times CompName$ with values $\{(comp_1, 'Context'), (comp_2, 'Sib'), \dots\}$, $Port = PortId \times PortName \times PortDir$ with values $\{(port_1, 'OutSrc', output), \dots\}$, or $Channel = ChanId \times ChanName$ with values $\{(channel_1, 'OutLeft'), \dots\}$; examples for association relations are $SubComp = CompId \times CompId$ with values $\{(comp_1, comp_2), \dots\}$ or $srcPort = ChanId \times PortId$ with values $\{(channel_1, port_1), \dots\}$. Intuitively, the conceptual domain describes the domain, from which specific instances of the description of an actual system are constructed.

Based on the conceptual domain, the conceptual model is the set of all possible system models that can be constructed within this domain. Intuitively, each system model is a “sub-model” of the conceptual domain, with sub-sets of its entities and relations. In order to be a proper system model, a subset of the conceptual domain generally must fulfill additional constraints; typical examples are the constraints in meta-models represented as class diagrams. In case of the AutoFOCUS conceptual model shown in Figure 2, e.g., each port or channel element must have an associated component in the `chanComp` and `portComp` relation, resp.; furthermore, each channel must have an associated source and destination port in the `srcPort` and `dstPort` relation.

2.2 Structure of the Model

The transformation framework provides mechanisms for a pure (i.e., side-effect free) declarative, rule-based approach to model transformation. To that end, the framework provides access to EMF Ecore-based models [14]. As described in Subsection 2.1, formally, a (conceptual) model is a collection of sets of elements (each described as a conceptual entity and its attribute values) and relations (each described as a pair of conceptual entities). To syntactically represent such a model, a Prolog term is used. Since these elements and relations are instances of classes and associations taken from an EMF Ecore model, the structure of the Prolog term – representing an instance of that

² For simplification purposes, the *Comment* attribute is ignored in the following.

model – is inferred from the structure of that model. The term comprises the classes and associations, of which the instance of the EMF Ecore model is constructed. It is grouped according to the structure of that model, depending on the package structure of the model and the classes and references of each package. The structure of the model is built using only simple elementary Prolog constructs, namely compound functor terms and list terms. Note that this construction obviously is not specific to Prolog; the same approach can be used for the embedding in other rule-based formalisms, e.g., functional languages like Haskell [15].

To access a model, the framework provides construction predicates to deconstruct and reconstruct a term representing a model. Since the structure makes only use of compound functor terms and list terms, only two classes of construction predicates are provided, namely the union operation and the composition operations.

Term Structure of the Model A model term describes the content of an instance of a EMF Ecore model. Each model term is a list of package terms, one for each packages of the EMF Ecore model. Each package term, in turn, describes the content of the package instance. It consists of a functor – identifying the package – with a sub-packages term, a classes terms, and an associations term as its argument. The sub-packages term describes the sub-packages of the package; it is a list of package terms.

The classes term describes the EClasses of the corresponding package. It is a list of class terms, one for each EClass of the package. Each class term consists of a functor – identifying the class - and an elements term. An elements term describes the collection of objects instantiating this class, and thus – in turn – is a list of element terms. Note that each elements term comprises only the collection of those objects of this class, which are not instantiations of subclasses of this class; objects instantiating specializations of this class are only contained in the elements terms corresponding to the most specific class. Finally, an element term - describing such an instance – consists of a functor – again identifying the class this object belongs to – with an entity identifying the element and attributes as arguments. Each of the attributes are atomic representations of the corresponding values of the attributes of the represented object. The entity is a regular atom, unique for each element term.

Similarly to the elements term, the associations terms describe the associations, i.e., the instances of the EReferences of the EClasses, for the corresponding package. Again, it is a list of association terms, with each association term consisting of a functor – identifying the association - and an relations term, describing the content of the association. The relations term is a list of relation terms, each term consisting of a functor – identifying the relation – and the entity identifiers of the related objects. In detail, the Prolog model term has the structure shown in Table 1 in the BNF notation with corresponding *non-terminals* and *terminals*.

The functors of the compound terms are deduced from the EMF Ecore model, which the model term is representing:

- the functor of a PackageTerm corresponds to the name of the EPackage the term is an instance of;
- the functor of a ClassTerm to the name of the EClass the term is an instance of; and finally

<i>ModelTerm</i>	::= [<i>PackageTerm</i> (, <i>PackageTerm</i>) [*]]
<i>PackageTerm</i>	::= <i>Functor</i> (<i>PackagesTerm</i> , <i>ClassesTerm</i> , <i>AssociationsTerm</i>)
<i>PackagesTerm</i>	::= [] [<i>PackageTerm</i> (, <i>PackageTerm</i>) [*]]
<i>ClassesTerm</i>	::= [] [<i>ClassTerm</i> (, <i>ClassTerm</i>) [*]]
<i>ClassTerm</i>	::= <i>Functor</i> (<i>ElementsTerm</i>)
<i>ElementsTerm</i>	::= [] [<i>ElementTerm</i> (, <i>ElementTerm</i>) [*]]
<i>ElementTerm</i>	::= <i>Functor</i> (<i>Entity</i> (, <i>AttributeValue</i>) [*])
<i>Entity</i>	::= <i>Atom</i>
<i>AttributeValue</i>	::= <i>Atom</i>
<i>AssociationsTerm</i>	::= [] [<i>AssociationTerm</i> (, <i>AssociationTerm</i>) [*]]
<i>AssociationTerm</i>	::= <i>Functor</i> (<i>RelationsTerm</i>)
<i>RelationsTerm</i>	::= [] [<i>RelationTerm</i> (, <i>RelationTerm</i>) [*]]
<i>RelationTerm</i>	::= <i>Functor</i> (<i>Entity</i> , <i>Entity</i>)

Table 1. The Prolog Structure of a Model Term

- the functor of an AssociationTerm corresponds to the name of the EReference the term in an instance of.

Since EMF – unlike MOF – does not support associations as first-class concepts like EClasses but uses EReferences instead, EReference names are not necessarily unique within a package. Therefore, if present, EAnnotation attributes of EReferences are used as the functors of an AssociationTerm. Similarly, the atoms of the attributes are deduced from the instance of the EMF Ecore model, which the model term is representing:

- the entity atom corresponds to the object identifier of an instance of a EClass, while
- the attribute corresponds to the attribute value of an instance of an EClass.

Currently, while basically also multi-valued attributes can be handled by the formalism, only single-valued attributes like references (including null references), basic types, and enumerations are supported by the implementation.

2.3 Construction Predicates

In a strictly declarative rule-based approach to model-transformation, the transformation is described in terms of a predicate, relating the models before and after the transformation. Therefore, mechanisms are needed in form of predicates to deconstruct a model into its parts as well as to construct a model from its parts. As the structure of the model is defined using only compound functor terms and list terms, only two forms of predicates are needed: union and composition operations.

List Construction The construction and deconstruction of lists is managed by means of the union predicate `union/3` with template³

³ According to standard convention, arbitrary/input/output argument of predicates are indicated by ?/+/-.

```
union(?Left,?Right,?All)
```

such that `union(Left,Right,All)` is true if all elements of list `All` are either elements of `Left` or `Right`, and vice versa. Thus, e.g., `union([1,3,5],R,[1,2,3,4,5])` succeeds with `R = [2,4]`.

Compound Construction Since the compound structures used to build the model instances depend on the actual structure of the EMF Ecore model, only the general schemata used are described. Depending on whether a package, class/element, or association/relation is described, different schemata are used. In all three schemata the name of the package, class, or relation is used as the name of the predicate for the compound construction.

Package Compounds The construction and deconstruction of packages is managed by means of package predicates of the form `package/4` with template

```
package(?Package,?Subpackages,?Classes,?Associations)
```

where `package` is the name of the package (de)constructed. Thus, e.g., a package named `structure` in the EMF Ecore model is represented by the compound constructor `structure`. The predicate is true if `Package` consists of subpackages `Subpackages`, classes `Classes`, and associations `Associations`. The predicate is generally used in the form `package(+Package,-Subpackages,-Classes,-Associations)` to deconstruct a compound structure into its constituents, and `package(-Package,+Subpackages,+Classes,+Associations)` to construct a compound structure from its constituents.

Class and Element Compounds The (de)construction of classes/elements is managed by means of class/element predicates of the form `class/2` and `class/N+2` where `N` is the number of the attributes of the corresponding class, with templates

```
class(?Class,?Elements)
class(?Element,?Entity,?Attribute1,...,?AttributeN)
```

where `class` is the name of the class and `element` (de)constructed. Thus, e.g., the class named `Component` in the EMF Ecore model in Figure 2 is represented by the compound constructor `Component`. The class predicate is true if `Class` is the list of `Objects`; it is generally used in the form `class(+Class,Objects)` to deconstruct a class into its list of objects, and `class(-Class,+Objects)` to construct a class from a list of objects. Similarly, the element predicate is true if `Element` is an `Entity` with attributes `Attribute1,...,AttributeN`; it can be used to deconstruct an element into its entity and attributes via `class(+Element,-Entity,-Attribute1,...,-AttributeN)`, to construct an element from an entity and attributes (e.g. to change the attributes of an element) via `class(-Element,+Entity,+Attribute1,...,AttributeN)`, or to construct an element including its entity from the attributes via `class(-Element,-Entity,+Attribute1,...,AttributeN)`. Thus,

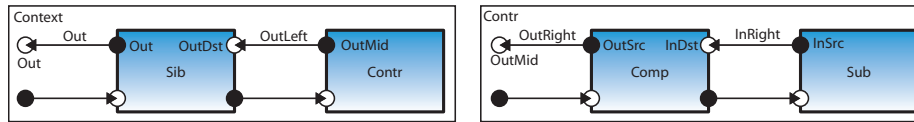


Fig. 3. Example: Result of Pushing Down Component Comp into Container Contr

e.g., `Component(Components, [Context, Sib, Contr])` is used to construct a class `Components` from a list of objects `Context`, `Sib`, and `Contr`. Similarly, `Component(Contr, Container, "Contr", "The container element")` is used to construct an element `Contr` with entity `Container`, name "Contr", and comment "The container element".

Association and Relation Compounds The construction and deconstruction of associations and relations is managed by means of association and relation predicate of the form `association/2` and `association/3` with templates

```
association(?Association, ?Relations)
association(?Relation, ?Entity1, ?Entity2)
```

where `association` is the name of the association and relation constructed/deconstructed. Thus, e.g., a relation named `subComponent` in the EMF Ecore model in Figure 2 is represented by the compound constructor `subComponent`. The relation predicate is true if `Association` is the list of `Relations`; it is generally used in the form `association(+Association, -Relations)` to deconstruct an association into its list of relations, and `association(-Association, +Relations)` to construct an association from a list of relations. Similarly, the relation predicate is true if `Relation` associates `Entity1` and `Entity2`; it is used to deconstruct a relation into its associated entities via `association(+Relation, -Entity1, -Entity2)` and to construct a relation between two entities via `association(-Relation, +Entity1, +Entity2)`. E.g., `subComponent(subComps, [CtxtSib, CtxtComp, CtxtContr])` is used to construct the subcomponent association `subComps` from the list of relations `CtxtSib`, `CtxtComp`, and `CtxtContr`. Similarly, `subComponent(CtxtContr, Context, Contr)` is used to construct relation `CtxtContr` with `Contr` being the subcomponent of `Context`.

3 Transformation Definition

The conceptual model and its structure defined in Section 2 was introduced to define transformations of system models as shown in Figure 1, in order to improve the development process by mechanized design steps. A typical transformation step is the pushing down of a component into a container component, making it a subcomponent of that container. Figure 3 shows the result of such a transformation pushing down component `Comp` of the system in Figure 1 into the container `Contr`. Besides changing the super component of `Comp` from `Context` to `Contr`, furthermore channel chains (e.g., `InLeft`,

InRight) from a port of a subcomponent (e.g., *Sub*) of the container (e.g., *InSrc*) via an intermediate port of the container (e.g., *InMid*) to a port of *Comp* (e.g., *InDst*) have to be shortened by eliminating the first channel and the intermediate port; symmetrically, a channel (e.g. *OutLeft*) from a port of the component (e.g., *OutSrc*) to a port (e.g., *OutDst*) of a sibling (e.g., *Sib*) of the component have to be extended by an additional channel (e.g., *OutRight*) and an intermediate port (e.g., *OutMid*).

In a relational approach to the description of model transformations, such a transformation is described as a relation between the model prior to the transformation (e.g., as given in Figure 1) and the model after the transformation (e.g., as given in Figure 3). In this section, the basic principles of describing transformations as relations are described in Subsection 3.1, while Subsection 3.2 shows how rules similar to graph grammars can be used as a specific description technique.

3.1 Transformations as Relations

In case of the push-down operation, the relation describing the transformation has the interface

```
pushpull (PreModel, Comp, Contr, PostModel)
```

with parameter *PreModel* for the model before the transformation, parameter *PostModel* for the model after the transformation, and parameters *Comp* and *Contr* for the component of the model to be pushed down and the component to contain the pushed-down component, respectively. In the relational approach presented here, a transformation is basically described by breaking down the pre-model into its constituents and build up the post-model from those constituents using the relations from Section 2, potentially adding or removing elements and relations.

With *PreModel* taken from the conceptual domain described in Figure 2 and packaged in a single package *structure* with no sub-packages, it can be decomposed in contained classes (e.g., *Port* and *Channel*) and associations (e.g., *portComp* and *srcPort*) via

```
structure(PreModel,PrePackages,PreClass,PreAssoc),
union([PreComps,PrePorts,PreChans],[],PreClass),
  channel(PreChans,InChans),port(PrePorts,InPorts),
union([PrePortComp,PreSrcPort],[],PreAssoc),
  portComp(PrePortComp,InPortComp),srcPort(PreSrcPort,InSrcPort)
```

In the same fashion, *PostModel* can be composed. Besides using the basic relations to construct and deconstruct models (and add or remove elements and relations, as shown in the next subsection), new relations can be defined to support a modular description of transformation, decomposing rules into sub-rules. E.g., in the *pushpull* relation, the transformation can be decomposed into the reallocation of the component and the rearrangement of the channel-chains; for the latter, then a sub-rule *splitmerge* is introduced, as explained in the next subsection.

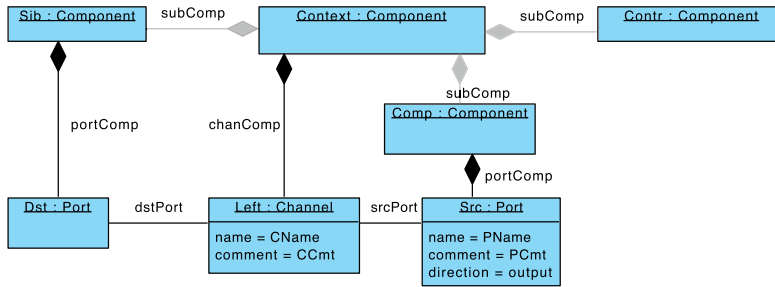


Fig. 4. Precondition of the Split-Merge-Transformation

3.2 Pattern-Like Rules

To define the transformation steps for splitting and merging channels during the push-down operation, we use a relation `splitmerge`, merging/splitting a channel starting or ending at a port of the pushed-down component `Comp`. This relation – formalized as a Prolog predicate `splitmerge` – with interface

```

splitmerge(
  InPorts, InChans, InPortComp, InChanComp, InSrcPort, InDstPort,
  Context, Comp, Contr,
  OutPorts, OutChans, OutPortComp, OutChanComp, OutSrcPort, OutDstPort)
  
```

relates the *Port* and *Channel* elements as well as the *PortComp*, *ChanComp*, *SrcPort*, and *DstPort* associations of the models before and after the transformation, depending on the component `Comp` to be relocated, its container component `Context`, and the component `Contr` it is pushed into. The splitting/merging of a channel connected to `Comp` depends on whether the channel is an input or an output channel of `Comp`, and whether it is connected to `Contr` or a sibling component within `Context`. Therefore, in a declarative approach, we introduce four different – recursive– split/merge rules for those four cases, each with the interface described above. Furthermore, a fifth rule is added, covering the case that no channels have to be split/merged, thus ensuring the termination of the recursive rule application.

To define the bodies of those rules, we use a pattern-like formalization, describing conditions over the models before and after the transformation step. These conditions are described in form of element and association terms that must be present in these models. Figures 4 and 5 give a graphical representation of these conditions in form of object diagrams – defining the transformation rule for splitting an output channel of `Comp` connected to a sibling component `Sib` – by describing the elements and relations that must be contained in the model before and after the transformation.⁴ Elements and relations present in the precondition in Figure 4 but not in the postcondition in Figure 5 are removed during the transformation, elements and relations present in the postcondition but not in the precondition are added during transformation, all others are left unchanged during the transformation.

⁴ Grayed-out associations are not used in the patterns but added only for clarification.

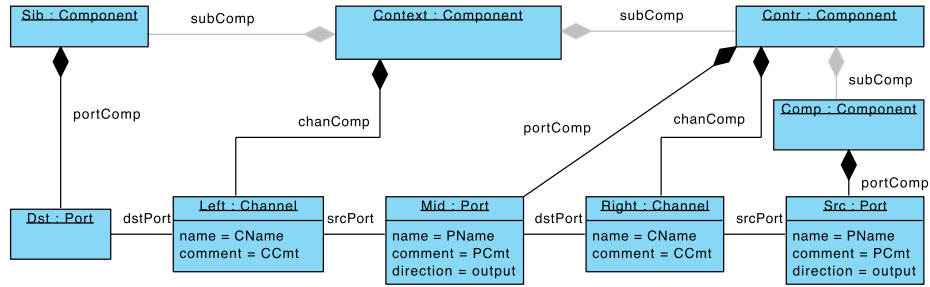


Fig. 5. Postcondition of the Split-Merge-Transformation

Figure 4 states that before the transformation, *Comp* and *Sib* each have an associated port *Src* and *Dst*, connected by channel *Left*, owned by *Context*. Figure 5 states that after the transformation, *Comp* is a subcomponent of *Contr*; furthermore, a port *Mid* of component *Contr* must be present with the same attributes as *Src*, as well as a channel *Right* owned by *Contr*, with *Left* connecting *Mid* and *Dst*, and *Right* connecting *Src* and *Mid*.

To define this transformation, the conceptual model and its structured representation introduced in Section 2 are used. Figure 6 shows the relational rule-based formalization of this step; underlined relations correspond to the construction predicates from Section 2.⁵ Note that this single rule combines the information of *both graphical representations* in Figures 4 and 5; it furthermore also describes the order in which individual split-merge steps are chained together in a modular fashion. Within a graphical specification of transformations, this later step cannot be described using those object-diagram-like diagrams at all. Therefore, graphical specifications require to use additional forms of diagrams, e.g., state-transition diagrams as used in [3], [4], or [5]. Unlike those, the approach here allows to simply pass information in form of parameters from the execution of one rule to the next.

Lines 1 to 7 describe the conditions before the transformation, and thus formalize the conditions described in Figure 4. Similarly, lines 11 to 18 describe the conditions after the transformation, thus formalizing the conditions described in Figure 5. Since the relational rule-based approach allows the use of recursively defined transformations, lines 8 to 10 introduce further transformation steps after removing the elements of the precondition and before introducing the elements of the postcondition. Lines 1 to 7 use a common scheme to ensure the validity of the conditions for the elements and relations taking part in the transformation: for each element and relation

- its occurrence in the list of corresponding elements/relations of the pre-model is established via the union constructor
- its contained entities and attributes within the pre-model are established via its corresponding functor

⁵ For ease of reading, functors for element relations from Figure 2 with capitalized identifiers like *Port* or *Channel* are written as *port* or *channel*; in the actual application, the versions '*Port*' and '*Channel*' are used.

```

1 union([SrcComp,DistSib],InPortComp,PrePortComp),
2   portComp(SrcComp,Src,Comp),portComp(DstSib,Dst,Sib),
3 union([LeftSrc],InSrcPort,PreSrcPort),srcPort(LeftSrc,Left,Src),
4 union([LeftDst],InDstPort,PreDstPort),dstPort(LeftDst,Left,Dst),
5 union([SrcPort],InPorts,PrePorts),port(SrcPort,Src,PName,PComment,output),
6 union([LeftChan],InChans,PreChans),channel(LeftChan,Left,CName,CComment),
7 union([LeftCxt],InChanComp,PreChanComp),chanComp(LeftCxt,Left,Context),
8 splitmerge(InPorts,InChans,InPortComp,InChanComp,InSrcPort,InDstPort,
9   Context,Comp,Contr,
10  OutPorts,OutChans,OutPortComp,OutChanComp,OutSrcPort,OutDstPort),
11 chanComp(RightCnt,Right,Contr),union([LeftCxt,RightCnt],OutChanComp,PostChanComp),
12 channel(RightChan,Right,CName,CComment),union([LeftChan,RightChan],OutChans,PostChans),
13 port(MidPort,Mid,PName,PComment,output),union([SrcPort,MidPort,DstPort],OutPorts,PostPorts),
14 dstPort(LeftMid,Left,Mid),dstPort(RightDst,Right,Dst),
15   union([LeftMid,RightDst],OutDstPort,PostDstPort).
16 srcPort(LeftSrc,Left,Src),srcPort(RightMid,Right,Mid),
17   union([LeftSrc,RightMid],OutSrcPort,PostSrcPort),
18 portComp(MidCnt,Mid,Contr),union([SrcComp,MidCnt,DstSib],OutPortComp,PostPortComp).

```

Fig. 6. A Split-Merge Rule for Sibling Channels

Thus, e.g., line 5 ensures the existence of an port element `SrcPort` consisting of port entity `Src` and attributes `PName`, `PComment`, and `output`, while line 7 ensures the existence of a `portComp` relation `SrcComp` linking port `Src` to component `Comp`. The union constructor is not only used to establish the occurrence of elements and relations in the pre-model; it is furthermore used to remove the identified elements and relations from the pre-model. E.g., line 5 also removes the port element `SrcPort` from the port class `PrePorts` of the pre-model before assigning the remainder to the port class `InPorts`, which is used in line 8 as input parameter of the recursive application of the `splitmerge` relation.

The scheme of lines 1 to 7 is used in symmetrical fashion in lines 11 to 18 to ensure the validity of the conditions for the elements and relations of the post-model. Thus, e.g., line 13 ensures that a port element `MidPort`, consisting of port entity `Mid` and attribute values defined by the variables `PName` and `PComment`, as well as the constant `output` of enumeration `Direction`, exists within port class `PostPorts`; similarly, line 11 ensures that a `chanComp` relation `RightCnt`, linking channel `Right` to component `Contr`, exists within `compChan` association `PostChanComp`, together with relation `LeftCxt`.

In a purely relational interpretation of declarative transformations, the ordering of the relations used to define a rule is irrelevant. Thus, using that kind of interpretation, a transformation can be applied in two ways. E.g., in case of the `pushpull`-transformation, the relation can be used to push-down a component into its container by assigning the pre- and post-models to the `PreModel` and `PostModel` parameter, respectively. Symmetrically it can be used to pull-up a component from a container by assigning the pre-model to the `PostModel` and the post-model to the `PreModel` parameter. As however the Prolog engine interpreting the declarative transformation rules

```

1 union([SrcComp,MidContr,DstSib],InPortComp,PrePortComp),portComp(SrcComp,Src,Comp),
2   portComp(MidContr,Mid,Contr),portComp(DstSib,Dst,Sib),
3 union([RightSrc,LeftMid],InSrcPort,PreSrcPort),
4   srcPort(RightSrc,Right,Src),srcPort(LeftMid,Left,Mid),
5 union([RightMid,LeftDst],InDstPort,PreDstPort),
6   dstPort(RightMid,Right,Mid),dstPort(LeftDst,Left,Dst),
7 union([SrcPort,MidPort,DstPort],InPorts,PrePorts),port(SrcPort,Src,--,output),
8   port(MidPort,Mid,--,output),
9 union([LeftChan,RightChan],InChans,PreChans),
10  channel(LeftChan,Left,--),channel(RightChan,Right,--),
11 union([LeftCxt,RightCnt],InChanComp,PreChanComp),chanComp(LeftCxt,Left,Context),
12  chanComp(RightCnt,Right,Contr),
13 splitmerge(InPorts,InChans,InPortComp,InChanComp,InSrcPort,InDstPort,
14  Context,Comp,Contr,
15  OutPorts,OutChans,OutPortComp,OutChanComp,OutSrcPort,OutDstPort),
16 union([LeftCxt],OutChanComp,PostChanComp),
17 union([LeftChan],OutChans,PostChans),
18 union([SrcPort],OutPorts,PostPorts),
19 union([LeftDst],OutDstPort,PostDstPort),
20 srcPort(LeftSrc,Left,Src),union([LeftSrc],OutSrcPort,PostSrcPort),
21 union([SrcComp,DstSib],OutPortComp,PostPortComp).

```

Fig. 7. A Split-Merge Rule for Container Channels

uses a sequential evaluation of the relations within a rule, the ordering becomes decisive concerning the *efficiency of the application* of the transformation. Thus, in the rules of Figures 6 and 7, the relations are ordered in correspondence with the binding of variables. E.g., in Figure 6 the binding extends from `Comp` over `Src` and `Left` to `PName`, `PComment`, `CName`, and `CComment`.

Figure 7 shows a variant form of the `splitmerge` relation.⁶ While the previous variant deals with the introduction of an additional channel connecting `Comp` to a sibling component `Sib` when pushing it down into `Contr`, this variant deals with the elimination of a channel connecting `Comp` to a subcomponent `Sub` of `Contr`. Again, lines 1 to 12 define the preconditions of the transformation, lines 16 to 21 define the postconditions, and lines 13 to 15 make use of the recursively defined `splitmerge` relation.

Since the two `splitmerge` rules are inverse transformation of each other, they can be essentially obtained by exchanging element and relation terms of the pre-models with those of the post-model, and the reordering of the clauses. Thus, e.g., the union-term of line 1 of Figure 6 corresponds to the union-term of line 21 of Figure 7 as well as the union-term of line 18 of the former corresponds to the union-term of line 1 of the latter, when swapping `InPortComp` and `PrePortComp` with `OutPortComp` and `PostPortComp`, resp. Similarly, the `chanComp`-functor terms in Figure 6 of lines 7 and 11 correspond to the `chanComp`-functor terms in Figure 7 of lines 16 and 12.

⁶ For ease of reading, the anonymous variable “_” is used in the position of irrelevant attributes.

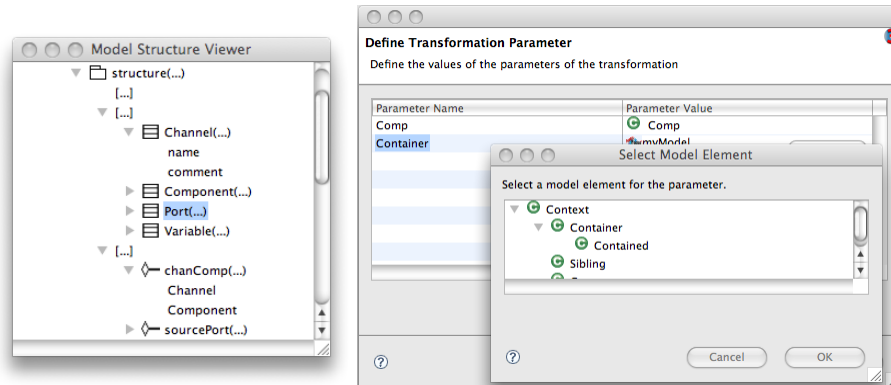


Fig. 8. Support for Defining and Applying Model Transformations

4 Tool Support

The presented approach has been implemented as an Eclipse plugin using the tuProlog engine [16]. While currently intended for the transformation of AutoFOCUS models, due to its generic nature it supports the transformation of EMF Ecore [14] models in general. As illustrated, the implemented plug-in provides tool support both for the definition of transformation and the transformation execution.

4.1 Transformation Definition

From the point of view of the designer of the transformation, its description consists of

- the declaration of the *Prolog clauses forming the rules* of the transformation relation,
- the declaration of the *root clause of the rule set*, and
- the declaration of the *list of parameters* of the root clause including their type (i.e., input model, output model, Boolean, Integer, String, or model entity),

provided in form of an XML file.

Since transformations depend on the structure of the model before and after the transformation, and transformations are intended to be designed at run-time of the AutoFOCUS framework, the designer of the model is provided with information on the structure of the conceptual model at runtime. As shown in the left-hand side of Figure 8, for that purpose, a model structure browser describing the term structure is provided as part of the online help documentation, based on the EMF Ecore reflection mechanism. By providing the term structure rather than the ECore structure, the declaration of the transformation rules is simplified.

The term structure is represented by providing the tree of relations used to construct the term as shown in Table 1; relations are E.g., as shown in Figure 8, the structure package term can be broken up into lists of subpackage, class, and association terms using the functor `structure`; similarly, the channel element term can be broken up into the attributes `name` and `comment` using the functor `Channel`.

4.2 Transformation Execution

The transformation is provided in form of a transformation wizard, available for AutoFOCUS projects and guiding the user through the three transformation steps

1. Selecting the transformation declaration
2. Defining the parameters of the transformation
3. Declaring the result model

plus an optional debug step. In the first step, the file containing the transformation description is selected. In the second step, as shown in the right-hand side of Figure 8, the parameters of the transformation rule – without the pre- and post model parameter – are defined. In the case of the push-pull transformation, suitable component entities must be assigned to the parameters *Comp* and *Container* for the to-be-pushed-down component and the container component. To define entity parameters, a model browser is provided to select suitable elements from the pre-model. After declaring, i.e., naming and locating, a model to contain the result of the transformation in the third step, the transformation can be executed. Optionally, the execution can be traced for debugging purposes, to analyse each application of each transformation rule.

5 Conclusion and Outlook

The AutoFOCUS transformation framework supports the transformation of EMF Ecore models using a declarative relational style. The framework provides a term-structure representation of the EMF Ecore description and offers a Prolog rule-based interpretation, and thus allows a simple, precise, and modular specification of transformation relations on the problem- rather than the implementation-level. By taking the operational aspects into consideration, the purely relational declarative form of specification can be tuned to ensure an efficient execution. Naturally, the abstraction from the object lattice of an EMF Ecore model to a model term of the conceptual model leads to some loss of efficiency – with the dereferencing of unique associations as the most prominent case (constant vs. linear complexity). However, in our experiments, the approach is practically feasible for real-world sized models (e.g, refactoring models consisting of more than 3000 elements and more than 5000 relations within a few seconds).

While the implementation has demonstrated the applicability of the approach, for a more thorough analysis a larger comparative case study considering the approaches mentioned in Subsection 1.1 is necessary, especially considered usability aspects like complexity of transformation specification and efficiency of executions. Furthermore, since the rule-based approach allows very general forms of application, other forms of application (e.g., view generation, transformations involving user interactions) will be integrated to extend the current implementation.

Finally, the current version of the framework covered those features of EMF that are sufficient to handle the complete AutoFOCUS conceptual model (including, e.g., data types and state machines). For other meta models like the UML 2.1 additional features (sorted) multi-valued attributes must be included, which – due to the structure- and list-based formalization – can be readily integrated in the current formalization.

Acknowledgements

It is a pleasure to thank Florian Hölzl, Benjamin Hummel, and Elmar Jürgens for their help in getting the implementation running, and Florian Hölzl and Peter Braun for the discussion on rule-based transformation.

References

- [1] Fowler, M., Scott, K.: UML Distilled. Addison-Wesley (1997)
- [2] Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: ESEC/FSE'07, ACM Press (2007)
- [3] Varro, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S., eds.: UML 2004, Springer (2004) LNCS 3273.
- [4] Grunske, L., Geiger, L., Lawley, M.: A graphical specification of model transformations with triple graph grammars. In Hartman, A., Kreische, D., eds.: Model Driven Architecture. Volume 3748 of LNCS., Springer (2005)
- [5] Sprinkle, J., Agrawal, A., Levendovszky, T.: Domain Model Translation Using Graph Transformations. In: ECBS2003 - Engineering of Computer-Based Systems. (2003)
- [6] Rozenberg, G., ed.: Handbook on Graph Grammars and Computing by Graph Transformation: Foundations. World Scientific (1997)
- [7] OMG: Initial submission to the MOF 2.0 Q/V/T RFP. Technical Report ad/03-03-27, Object Management Group (OMG), <http://www.omg.org> (2003)
- [8] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: OOPSLA '06, ACM Press (2006) 719–720
- [9] Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. In: Graph Transformation. Volume 2505 of LNCS. (2002)
- [10] Lawley, M., Steel, J.: Practical declarative model transformation with tekat. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of LNCS., Springer (2006)
- [11] White, J., Schmidt, D.C., Nechypurenko, A., Wuchner, E.: Introduction to the generic eclipse modelling system
- [12] Varró, G., Friedl, K., Varró, D.: Implementing a graph transformation engine in relational databases. *Software and Systems Modeling* **5** (2006)
- [13] Schätz, B., Huber, F.: Integrating formal description techniques. In Wing, J.M., Woodcock, J., Davies, J., eds.: FM'99, Springer Verlag (1999) LNCS 1709.
- [14] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison Wesley Professional (2007) Second Edition.
- [15] Jones, S.P.: Haskell 98 language and libraries: the Revised Report. Cambridge University Press (2003)
- [16] Denti, E., Omicini, A., Ricci, A.: Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming* **57**(2) (2005) 217–250