Electronic Communications of the EASST
Volume 29 (2010)

Proceedings of the Ninth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010)

Verification of Model Transformations

Bernhard Schätz

13 pages

Guest Editors: Jochen Küster, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Verification of Model Transformations

Bernhard Schätz

fortiss GmbH
Guerickestr. 25, 80805 Mnchen, Germany
schaetz@fortiss.org

Abstract: With the increasing use of automatic transformations of models, the correctness of these transformations becomes an increasingly important issue. Especially for model transformation generally defined using abstract description techniques like graph transformations or declarative relational specifications, however, establishing the soundness of those transformations by test-based approaches is not straight-forward. We show how formal verification of soundness conditions over such declarative relational style transformations can be performed using an interactive theorem prover. The relational style allows a direct translation of transformations as well as associated soundness conditions into corresponding axioms and theorems. Using the Isabelle theorem prover, the approach is demonstrated for a refactoring transformation and a connectedness soundness condition.

Keywords: Model transformation, rule-based, verification, theorem prover

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. Besides increasing efficiency, these transformations can offer consistency ensuring modification of models, ranging from refactoring steps to improve the architecture of a system to the consistent integration of standard behavior. However, with the increased use of transformation, the question of the correctness of transformations arises: How can we verify that the models constructed via transformation are 'well-formed' given a 'well-formed' source model, e.g., by ensuring that no relevant elements of the source model are absent in the target model. Obviously, testing is one possible way of ensuring the correctness of transformations. However, concepts like coverage etc. are not immediately transferable to model-transformations, especially if those are rules-based or declarative.

In the following, a approach for the verification of transformations is introduced, supporting the formal proof of properties over these transformations. The approach uses a declarative relational style to provide a transformation mechanism, implemented on the Eclipse/EMF Ecore platform, using a Prolog rule-based interpretation.

1.1 Related Approaches

Verification of model transformations has been specifically investigated for graph-based transformation techniques (e.g., [GGL+06] and [Str08]). In that respect, the presented approach is similar: 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. A theorem prover based on on high-order logics is used to prove characteristics of the transformation by deducing properties of the target model from some properties of the source model.

In contrast to other graph-based approaches like MOFLON/TGG [KKS07], Viatra [VP04], or FuJaBa [GGL05], however, here the specification of transformations is not based on triple graph-grammars or graphical, rule-based descriptions, but uses a textual description based on a relational, declarative calculus. Therefore, in contrast to those approaches, the approach introduced here uses only a single formalism to describe basic transformations as well as their compositions.Furthermore, 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. Thus, a specification can be immediately used for verification without complex translations; furthermore, proofs on the formal level more directly reflect intuitive reasoning about the transformation.

This homogeneity is especially important for verification since is drastically simplifies the construction of proofs: [GGL+06] focusses on TGG-based translation and therefore uses substantial proof parts to model (and verify) the effective construction of correspondence graphs to describe the application of individual graph rules. Furthermore – due to that approach – there structural induction over the pre/post-models is used which is less convenient when if non-translation transformations are verified. Here, in contrast, induction over the transformation itself rather than the pre/post models is performed, thus having a more direct proof principle and avoiding the proof overhead of correspondence graphs and applicability conditions. Similarly, [Str08], – using *Isabelle* as theorem proving support, too – also requires substantial effort to specify and verify correspondence graph and application conditions for single transformation rules as well their combination using *while* and *case* constructs. Thus, the application and ordering of rules provided *implicitly* by a TGG approach has to be verified *explicitly* and using rather different proof principles. In contrast, here, a more direct and homogenous form of proof is supported by the declarative rule-based style. [CR09] uses a relational description of graphs similar to the presented approach as well as similar proof principles. However, neither is their formalism is supported by an executable implementation nor do they use mechanic proof support.

a rule-based description of transformations.

Another advantage of the presented approach is its capability to interpret loose characterizations of the resulting model, supporting the exploration of a set of possible solutions. By making use of the back-tracking mechanism provided by Prolog, alternative transformation results can not only be applied to automatically search for an optimized solution, e.g., balanced component hierarchies, using guiding metrics; the set of possible solutions can also be incrementally generated to allow the user to interactively identify and select the appropriate solution.

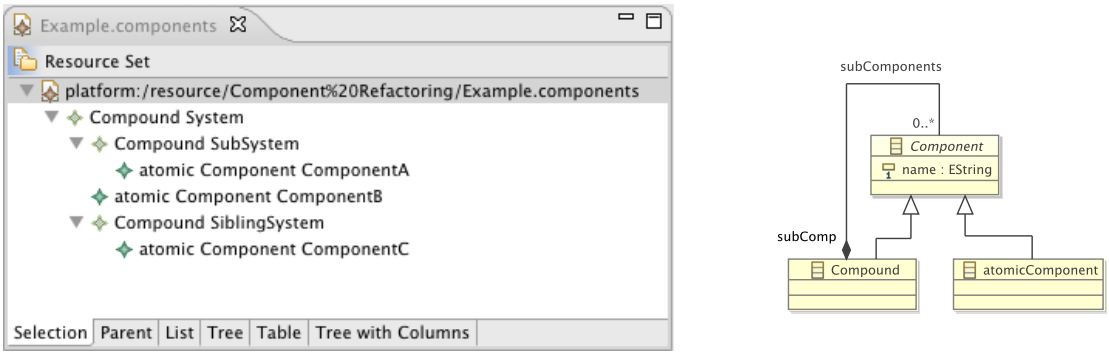

Figure 1: Example of Hierarchical Component Model and Corresponding Conceptual Model

1.2 Overview and Contribution

As the main contribution, an approach to *formally verify model transformations* is presented in the following sections. The approach is based on a transformation of EMF Ecore models using a completely *declarative relational* style in a *rule-based* fashion, introduced in [Sch08]. To provide such a form of transformations, the approach uses a term-based formalization of an EMF model as shown in Section 2. With this form of model representation, as shown in Section 3 transformations can be described as declarative relations in Prolog style, supporting rules similar to graph grammars as a specific description style.

Based on these previously established results, as *new contribution* in Section 4 the suitability of this declarative relational style of defining models and transformation rules for the verification of transformations is shown: The formalization of (meta-)models and transformation rules can be directly translated in representations suited for theorem provers for predicate logic like *Isabelle*; furthermore, due to the relational style correctness proofs of transformations can be performed by reasoning on the level of their specifications. Section 5 highlights some benefits and open issues.

2 Model Structure

To provide verified transformations of descriptions of systems, first the means of specifying a system in form of a system model is needed. The left-hand side of Figure 1 shows such a model, describing the hierarchical structure of the components of a system: the system System, consisting of subcomponents SubSystem, ComponentB, and SiblingSystem, the first and the last with subcomponents ComponentA and ComponentC, resp.[1]

To construct formalized descriptions of a system under development, a 'syntactic vocabulary' is needed. This conceptual model[2] characterizes all possible system models built from the *modeling concepts and their relations* used to construct a description of a system; typically, class

[1] For simplification, here only components and their containment-relation is modeled; other typical aspects like interfaces or communication links are ignored.

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

diagrams are used to describe them. The right-hand side of Figure 1 shows the corresponding conceptual model – with the concept of a Component with an attribute name and a subComp-relation – used to describe the architectural structure of a system.

2.1 Structure of the Model

The transformation framework provides mechanisms for a pure (i.e., side-effect free) declarative, rule-based approach to model transformation, accessing EMF Ecore-based models [SBPM07]. Based on the conceptual model, a system model consists of sets of elements (each described as a conceptual entity and its attribute values) and relations (each described as a pair of conceptual entities), syntactically represented as a Prolog term. 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 model – is inferred from the structure of that model. The structure of the model is built using only simple elementary Prolog constructs, namely compound functor terms and list terms. To access a model, the framework provides predicates to deconstruct and reconstruct a term representing a model. [Sch08] describes the model in more detail.

A *model term* describes 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. 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 is a list of element terms. Finally, an *element term* consists of a functor, 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 an elements term, each *associations term* describes 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 *relation term* consisting of a functor, identifying the relation, and the entity identificators 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*.[3]

The functors of the compound terms are deduced from the EMF Ecore model: The functor of a PackageTerm from the name of the EPackage; the functor of a ClassTerm from the name of the EClass; the functor of an AssociationTerm from the name of the EReference. 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 identificator of an instance of a EClass, while the attribute corresponds to the attribute value of an instance of an EClass.

[3] While actually a *ModelTerm* consists of a set of *PackageTerms*, here for simplification purposes only one *PackageTerm* is assumed.

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

Table 1: The Prolog Structure of a Model Term

2.2 Construction Predicates

In a strictly declarative rule-based approach, 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.

2.2.1 List Construction

The(de)construction of lists is managed by means of the union predicate `union/3` with template[4] `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]`.

2.2.2 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. In all three schemata – package, class/element, or association/relation – the name of the package, class, or relation is used as the name of the predicate for the compound construction.

Packages For (de)construction of packages, package predicates of the form `package/4` are used with template `package(?Package,?Subpackages, ?Classes,?Associations)` where package is the name of the package (de)constructed. Thus, e.g., a package named Architecture in the EMF Ecore model is represented by the compound constructor `Architecture`. The predicate is true if `Package` consists of subpackages `Subpackages`, classes `Classes`, and associations `Associations`.

[4] According to standard convention, arbitrary/input/output arguments of predicates are indicated by `?`/`+`/`-`.

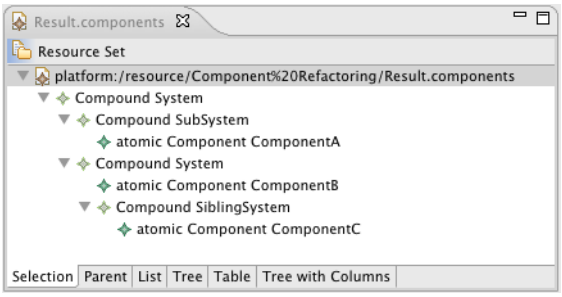

Figure 2: Example: Result of Clustering ComponentB and SiblingSystem

Classes and Elements For (de)construction of – non-abstract – classes/elements, class/element predicates of the form `class/2` and `class/N+2` are used where N is the number of the attributes of the corresponding class, with templates `class(?Class, ?Elements)` and `class(?Element,?Entity,?Attr1,...,?AttrN)` where class is the name of the class and element (de)constructed. Thus, e.g., the class named Compound in the EMF Ecore model in Figure 1 is represented by the compound constructor `Compound`. The class predicate is true if `Class` is the list of `Objects`; it is used to deconstruct a class into its list of objects, and vice versa. Similarly, the element predicate is true if `Element` is an `Entity` with attributes `Attr1`,...,`AttrN`; it can be used to deconstruct an element into its entity and attributes, to construct an element from an entity and attributes (e.g. to change the attributes of an element), or to construct a new element including its entity from the attributes. Thus, e.g., `Compound(Compounds,[Sys,Sub,Sib])` is used to construct a class `Compounds` from a list of objects `Sys`, `Sub`, and `Sib`. Similarly, `Compound(Sub,Subsys,"SubSystem")` is used to construct a new element `Sub` with entity `Subsys`, and name `"SubSystem"`.

Association and Relation Compounds For (de)construction of associations and relations, association and relation predicate of the form `association/2` and `association/3` are used with templates `association(?Association,?Relations)` and `association(?Relation,?Entity1,?Entity2)` where association is the name of the association and relation constructed/deconstructed. Thus, e.g., a relation named subComp in the EMF Ecore model in Figure 1 is represented by the compound constructor `subComp`. The relation predicate is true if `Association` is the list of `Relations`; it is generally used to deconstruct an association into its list of relations, and vice versa. Similarly, the relation predicate is true if `Relation` associates `Entity1` and `Entity2`; it is used to deconstruct a relation into its associated entities and vice versa. E.g., `subComp(subComps,[SubSys,SibSys])` is used to construct the subcomponent association `subComps` from the list of relations `SubSys` and `SibSys`. Similarly, `subComp(SubSys,Sub,Sys)` is used to construct relation `SubSys` with `Sub` being the subcomponent of `Sys`.

```
1  cluster(Pre,Group,Post) :−
2      Architecture(Pre,Pack,PreClass,PreAssoc),
3      Compound(PreComp,PreComps),union(OtherClass,[PreComp],PreClass),
4      subComp(PreSub,PreSubs),union(OtherAssoc,[PreSub],PreAssoc),
5      link(PreSubs,Group,PreRoot,OutSubs),
6      Compound(PreRootComp,PreRoot,Name),union([PreRootComp],Comps,PreComps),
7      subComp(NewSub,PostRoot,PreRoot),union([NewSub],OutSubs,InSubs),
8      Compound(PostRootComp,PostRoot,Name),union([PreRootComp,PostRootComp],Comps,PostComps),
9      link(PostSubs,Group,PostRoot,InSubs),
10     subComp(PostSub,PostSubs),union(OtherAssoc,[PostSub],PostAssoc),
11     Compound(PostComp,PostComps),union(OtherClass,[PostComp],PostClass),
12     Architecture(Post,Pack,PostClass,PostAssoc).
```

Figure 3: Cluster-Transformation: Rule for (De-)Constructing the Model

3 Transformation Definition

The conceptual model and its structure defined in Section 2 was introduced to define transformations of system models as shown in the left-hand side of Figure 1. A typical transformation step is the clustering of a group of sibling components within a container component, making them subcomponents of that container. Figure 2 shows the result of such a transformation clustering subcomponents ComponentB and SiblingSystem of component System in Figure 1 into a new System container. Besides introducing the new additional component System and making it a subcomponent of the original System root component, the transformation also requires changing the supercomponent of ComponentB and SiblingSystem.

In a relational approach to model transformations, such a transformation is described as a relation between the model prior to the transformation (e.g., as given in the left-hand side of Figure 1) and the model after the transformation (e.g., as given in Figure 2). In this section, the basic principles of describing transformations as relations are described.

3.1 Transformations as Relations

In case of the clustering operation, the relation describing the transformation has the interface `cluster(Pre,Group,Post)` with parameter `Pre` for the model before the transformation, parameter `Post` for the model after the transformation, and parameter `Group` for the group of components of the model to be clustered. 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 `Pre` taken from the conceptual domain described in Figure 1 and packaged in a single package *Architecture* with no sub-packages, it can be decomposed in contained classes (e.g., *Compound*) and associations (e.g., *subComp*) as shown in Figure 3, lines 2 to 4.[5] In the same fashion, `Post` can be composed in lines 12 to 10. Lines 6 to 8 obtain the *Name* of the common super-component with entity *PreRoot* of the group (line 6), provide

[5] For ease of reading, quotes required in Prolog for capital functor identifiers like *Architecture* or *Compound* are dropped.

```
1 link(Subs,[],Root,Subs).
2 link(InSubs,Group,Root,OutSubs) :−
3     subComp(SubRel,Sub,Root),union([Sub],Rest,Group),union([SubRel],Subs,InSubs),
4     link(Subs,Rest,Root,OutSubs).
```

Figure 4: Cluster-Transformation: Rule for (Un-)Linking SubComponents

a newly created compound container component *PostRootComp* this *Name* and entity *PostRoot* (line 8), and make this *PreRoot* the super-component of *PostRoot* (line 7). Note that the relation is bidirectional: Besides clustering a group of siblings into a common container, it can also be used to uncluster the group of subcomponents contained in a common container.

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 `cluster` relation, the transformation can be decomposed into the addition of the new container component and the reallocation of the components to be clustered; for the latter, then a sub-relation `link` with corresponding rules is introduced, as shown in Figure 4. Note that `link` is effectively used in both directions in the `cluster` relation: In line 5, `link` is used to unlink subcomponents by removing the `subComp`-associations between *Group* elements and the original component *PreRoot* from *PreSubs* to obtain *OutSubs*; in line 9, `link` is used to link subcomponents by adding the `subComp`-associations between *Group* elements and the new component *PostRoot* to *InSubs* to obtain *PostSubs*.

3.2 Transformations as Rules

To define the transformation steps for (un)linking components and subcomponents, relation `link(InSubs,Group,Root,OutSubs)` is used, by making the set *OutSubs* of associations the reduction of set *InSubs* when removing all `subComp`-associations between elements from *Group* and *Root*. The (un)linking of a group depends on whether the group is empty or not. Therefore, in a declarative approach, two different – recursive – (un)link rules for those two cases are needed, each with the interface described above.

To define these rules as shown in Figure 4, the conceptual model and its structured representation introduced in Section 2 are used. Line 1 simply states that in case of an empty group the sets of associations are the same since no elements can be (un)linked. This case also handles the termination of the inductive rule definition. In case of a non-empty group, line 3 (un)links a *Sub* element from the *Group* – leaving a rest *Rest* – and *Root*, while line 4 repeats this (un)linking recursively for the *Rest* of the group. Note that this rule-based description allows to compose complex transformations by simple application of rules in the body of another rule (like `link` in `cluster`). In contrast, graphical specifications generally use additional forms of diagrams, e.g., state-transition diagrams. As shown in the following section, this direct combination of rules, however, is essential to simplify the formal verification of the correctness of transformations.

4 Verification

The relational and declarative approach introduced in the previous sections supports an easy transition to formal reasoning. In this section, the formalization of an EMF Ecore meta-model in constructive type theory is presented, as well as the straight-forward formalization of transformations. Based on these formalizations, the construction of formal correctness proof is demonstrated using the example of typical invariants. To support formal verification, the interactive theorem prover *Isabelle/HOL* [NPW02] is applied.

4.1 Meta-Model Formalization

Isabelle/HOL supports the form of (typed) terms used to represent the EMF models in the rule-based transformation process. Thus, the transition from the specifications used in Section 2 to *Isabelle/HOL* is straight-forward, as shown in the – syntactically slightly simplified – formalization of the meta-model of Figure 1:[6]

```
1 typedecl ids
2 typedecl string
3 datatype comp = Comp ids string, atom = Atom ids string
4 datatype subComp = SubComp ids ids
5 datatype cls = Comp comp set | Atom atom set
6 datatype asc = SubComp subComp set
7 datatype architecture = Architecture cls set asc set
```

After introducing – via `typedecl` – uninterpreted *ids* and *string* types for representing entities and string attributes in lines 1 and 2, the corresponding element (line 3), relation (line 4), class (line 5), association (line 6), and package (line 7) term types are introduced simply by providing – via `datatype` – constructor functions, using the same scheme as introduced in Subsection 2.1.[7] Based on these constructors and using the set operations provided by *Isabelle/HOL*, Prolog model terms can be directly translated, thus enabling the translation of transformations.

4.2 Transformation Formalization

Besides type terms, *Isabelle* also supports the definition of predicates in a rule-based fashion analogue to the Prolog-based rules in the transformation approach. To define the transformation relations in *Isabelle*, *inductive* definitions of predicates are used to allow recursive definitions. The non-recursive `cluster` relation of Section 3 is – trivially inductively – defined via:[8]

```
1 inductive cluster :: architecture => ids set => model => bool where
2     pre = (Architecture preclass preassoc) &
3     precomp = (Comp precomps) & otherclass Un {precomp} = preclass &
4     presub = (SubComp presubs) & otherassoc Un {presub} = preassoc &
5     (link presubs group preroot outsubs) &
6     prerootcomp = (Comp preroot name) & {prerootcomp} Un comps = precomps &
7     newsub = (SubComp postroot preroot) & {newsub} Un outsubs = insubs &
```

[6] *set* introduces a set type, | a variant type, => a function type.

[7] The Compound and AtomicComponent element/class constructors are abbreviated to Comp and Atom, resp.

[8] Standard *Isabelle* notation is used, including `&`, `|`, and `-->` for conjunction, disjunction, and implication; `<=` and `:` for the subset and element relation; `?` for the existential quantor.

```
8   postrootcomp = (Comp postroot name) & {prerootcomp,postrootcomp} Un comps = postcomps &
9   (link postsubs group postroot insubs) &
10  postsub = (SubComp postsubs) & otherassoc Un {postsub} = postassoc &
11  postcomp = (Comp postcomps) & otherclass Un {postcomp} = postclass &
12  post = (Model postclass postassoc)
13      −−> (cluster pre group post)
```

Obviously, again the transition from the specifications used in the previous sections to *Isabelle/HOL* is straight-forward: Line 2 to 12 directly correspond to line 2 to line 12 in Figure 3; in the former only a direct formalization with equality combined the constructors and set union is used, while the later uses (de)construction predicates. Line 13 of the former corresponds to line 1 of the later. Line 1 additionally defines the type of the predicate in *Isabelle/HOL* designated by "::". In a similar fashion, the specification of `link` can be directly translated:

```
1  inductive link :: subComp set => ids set => ids => subComp set => bool where
2    (link subs {} root subs) |
3    (link subs rest root outsubs) −−> (link ({subComp.SubComp sub root} Un subs) ({sub} Un rest) root outsubs)
```

4.3 Proof Construction

Using the formalization of the transformations introduced above, now correctness properties of the clustering operation can be defined. In the following, two conditions – one concerning class and one concerning association properties – are considered:

1. Each Compound element contained in the pre-model is also contained in the post model.

2. Each subComp relation between a component and some super-component in the pre-model has a counter-part in the post-model for the same component and some – potentially different – super-component.

The first property is formalized as `theorem` *keep_Comp_cluster*:

```
1  theorem keep_Comp_cluster:
2    (cluster pre group post) & pre = (Architecture preclass preassoc) & post = (Architecture postclass postassoc) &
3    preComp = (Comp preComps) & preAtom = (Atom preAtoms) & {preComp, preAtom} = preclass &
4    postComp = (Comp postComps) & postAtom = (Atom postAtoms) & {postComp, postAtom} = postclass &
5    (somecomp:preComps) −−> (somecomp:postComps)
```

This theorem is straightforward to prove, requiring no induction but only case distinction. Therefore, the proof is mainly performed by `apply`ing *Isabelle*'s automatic proof tactics (e.g., *auto*, *clarify*, *clarsimp*), rendering the theorem (or lemma) applicable in further proof steps:

```
1  apply auto
2  apply (erule pushpull.cases)
3  apply clarify
4  apply (drule equalityD1)
5  apply (drule equalityD2)
6  apply (drule Un_sub_D)
7  apply (drule Un_sub_D)
8  apply clarsimp
```

Beside the case distinction (line 2), the proof requires three standard simplifications (lines 1, 3, 8) and four simple interactions deadline with equality and sub-set relation properties, where the latter could also be further automized by providing suitable rules.

The second, more challenging property is formalized as theorem *keep_subComp_cluster*:

```
1 theorem keep_subComp_cluster:
2   (cluster pre group post) & pre = (Architecture preclass preassoc) & post = (Architecture postclass postassoc) &
3   preSubComp = (SubComp preSubComps) & {preSubComp} = preassoc &
4   postSubComp = (SubComp postSubComps) & {postSubComp} = postassoc &
5   (? root. (SubComp some root):preSubComps) −−> (? root .(SubComp some root):postSubComps)
```

The proof script for theorem *keep_subComp_cluster* uses the same steps as before; however, since the corresponding super-component in a subComp-relation in the post-model is different whether the sub-component is in the group to be clustered or not, the proof requires one additional step – a lemma application – for distinction between these cases. To that end, corresponding lemmata are introduced and proved, e.g., *keep_link_group* to deal with the case on non-group elements. Since this distinction essentially affects `link`, these lemmata operate on the `link` relation:

```
1 lemma keep_link_group: (link pre group old lsubs) & (link post group new rsubs) −−> (lsubs <= rsubs & some:group)
2   −−> (SubComp some root):pre −−> (SubComp some root):post
```

Since these lemmata make use of the inductively defined relation `link`, induction must be used. However, besides suggesting the use of the induction principle on the definition of `link`, again the proof can performed fully automatic. These lemmata can be combined in a single lemma *keep_link* with a trivial proof:

```
1 lemma keep_link: (link pre group old lsubs) & (link post group new rsubs) −−> lsubs <= rsubs
2   −−> (? root. (SubComp some root):pre) −−> (? root. (SubComp some root):post)
```

In its proof, proven lemmata like *keep_link_group* can be applied in the form

```
1 apply (insert keep_link_group [of pre group old lsubs post new rsubs some])
```

The complete proof of theorem *keep_subComp_cluster* consists of the proof of the lemmata with 23 steps and 10 steps for the proof of the theorem itself with the resulting *keep_link* lemma.

5 Conclusion and Outlook

The PETE transformation framework – provided as an Eclipse PlugIn [Sch09] – supports the transformation of EMF Ecore models using a declarative relational style and allows a simple, precise, and modular specification of transformation relations on the problem- rather than the implementation-level. By including operational aspects, the relational declarative form of specification can be tuned to ensure an efficient execution. In the application to problem from the embedded software domain, the approach has demonstrated practical feasibility for medium real-world sized models (e.g, refactoring models consisting of more than 3000 elements and more than 5000 relations within a few seconds). Furthermore, debugging on the level of the specification supports the construction of transformations.

The use of a declarative relational style of specifying transformations is an important asset for the formal verification of correctness conditions of these transformations: It allows the direct translation of the conceptual model as well as the transformation rules into a predicate-logical

formalization. Since no indirections are introduced between the specifications on the execution and the verification level, the proof can be constructed following a natural argumentation. Using a verification tool like *Isabelle/HOL*, the verification process can be automized to a large extent.

While the previous sections have demonstrated the applicability of the approach, additional means of automation should be provided for a extensive application. This includes the mechanic translation of EMF Ecore models into the corresponding type definitions. Furthermore, the translation should include the definition of the basic manipulation predicates in the (de)constructor format to allow the 1:1 use of the executable specification of transformations in the verification. Additionally, general lemmata, tailor-made tactics, or using *ISAR* for more readable proof scripts should be provided to simplify proofs. Also, other property languages like OCL and pre/post schemata should be included, to circumvent the specification of property conditions on the level of predicate logics. Finally, the practicability of the verification approach requires the analysis of larger case studies to understand in which cases a formal verification of a rule-based transformation can be favorable, e.g., over a testing-based verification of a programming-level transformation.

Since the declarative relational style can also be used to support a search-based design-space exploration involving backtracking – e.g., when computing correct deployments in embedded systems – making test-based verification even more complex, simple formal verifiability of the correctness of such explorative transformations is especially helpful.

Bibliography

[CR09] S. A. da Costa, L. Ribeiro. Formal Verification of Graph Grammars using Mathematical Induction. *Electronic Notes in Theoretical Computer Science* 240:43–60, 2009.

[GGL05] L. Grunske, L. Geiger, M. Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In Hartman and Kreische (eds.), *Model Driven Architecture*. LNCS 3748. Springer, 2005.

[GGL+06] H. Giese, S. Glesner, J. Leitner, W. Schfer, R. Wagner. Towards Verified Model Transformations. In *In Proceedings of MoDeVa workshop associated to MoDELS'06*. Pp. 78–93. 2006.

[KKS07] F. Klar, A. Königs, A. Schürr. Model Transformation in the Large. In *ESEC/FSE'07*. ACM Press, 2007.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283. Springer, 2002.

[SBPM07] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2007. Second Edition.

[Sch08] B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. In Dragan Gasevic (ed.), *Software Language Engineering*. LNCS. Springer, 2008.

[Sch09] B. Schätz. Prolog EMF Transformation Eclipse-PlugIn. `www4.in.tum.de/~schaetz/PETE`, 2009.

[Str08] M. Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *Electr. Notes Theor. Comput. Sci.* 203(1):135–148, 2008.

[VP04] D. Varro, A. Pataricza. Generic and meta-transformations for model transformation engineering. In Baar et al. (eds.), *UML 2004*. Springer, 2004. LNCS 3273.