

Saying Hello World with PETE - A Solution to the TTC 2011 Instructive Case

Bernhard Schätz

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

PETE – Prolog EMF Transformation Engine – supports a declarative relational style based on the interpretation of a model as a structured term. It provides a transformation mechanism for the specification of transformations for the EMF Ecore platform, using a Prolog rule-based mechanism. PETE provides a basic declarative relational style as a core formalism, more suited for formal verification than pragmatic application. Language layers can be constructed using PETE, to improve conciseness of specifications. This approach is demonstrated in the context of the TTC 2011 Hello World example to provide a more pattern-based style along the lines of graph-based add/delete-patterns.

1 Motivation

PETE provides an experimentation platform to develop and evaluate transformation specification techniques, using a minimum core formalism with maximum expressiveness and precise semantics. This contribution demonstrates the application of PETE to language design by showing the definition of a new language layer on top to the core language, and its application to the HelloWorld case of [1].

The core formalism – in basic declarative relational style – corresponds to a relation-based mathematical style of formalizing transformations. While this style makes it well suited, e.g., for formal verification of transformations via theorem proving, it leads to rather verbose specifications of transformations. Therefore, from a pragmatic point it is not well-suited for the specification of real-world transformations. Here, more pattern-based approaches using similar concepts found in graph transformation, support a more condensed style. Thus, additionally to the core language, here a *pattern-based language extension* is introduced, which itself is formalized in terms of the core formalism. The translation between the two language layers is in turn implemented as interpreter with few Prolog rules.

In the remainder of the submission, in Section 2 the core language is introduced. In Section 3, the application of the core language to model transformations is explained, either directly via the declarative relational style or indirectly by adding the pattern-rule-formalism as an additional language layer. A short discussion of the pros and cons concludes, followed by the listings of the transformations of the HelloWorld case in either style.

2 PETE – Core Formalism

To construct descriptions of a system, a ‘syntactic vocabulary’ 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.

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

<i>ModelTerm</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

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 [6]. Based on the conceptual model, a PETE 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), 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: compound functor terms and list terms. To access a model, the framework provides predicates to deconstruct and reconstruct a term representing a model. [2] 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 package 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 term, 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 a 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 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*.²

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;

²While actually a *ModelTerm* consists of a set of *PackageTerms*, here for simplification purposes only one *PackageTerm* is assumed.

```

1 helloworldext(
2   [],
3   [Greeting([Greeting(Entity1)]),
4     GreetingMessage([GreetingMessage(Entity2, 'Hello World')]),
5     Person([Person(Entity3, 'TTC Participant')]),
6   [greetingMessage([greetingMessage(Entity1, Entity2)]),
7     person([person(Entity2, Entity3)])]

```

Table 2: The Prolog Structure of the Hello World Example

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 identifier of an instance of an EClass, while the attribute corresponds to the attribute value of an instance of an EClass.

Table 2 shows the representation of the *HelloWorldExt* example from the Live Contest.³

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.

The (de)construction of lists is managed by means of the union predicate `union/3` with template⁴ `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.

To simplify the use of sorted lists, the union operator also accepts unsorted arguments. These arguments are defined with curly braces, e.g., `{1, 3, 2, 5}`, provided standardly in Prolog.

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, in the *hello2* example in Section A.1, the *helloworldext* package is constructed by the compound constructor `helloworldext` in line 8.

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. The class predicate is true if `Class` is the list of `Elements`; 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

³For reason of readability, the effective representation of the entity terms is not shown.

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

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. In the example of *hello2* again, in line 2 the first *GreetingMessage* is used to construct a new element *GreetingMessage*, while the second is used to construct the class.

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 (de)constructed. 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. Again, in example *hello2* in line 6, the first *person* constructor is used to create a relation between two entities, while the second is used to construct the association.

3 Transformation Definition

The conceptual model and its structure defined in Section 2 was introduced to define transformations of system models. In a relational approach to model transformations, a transformation – like the migration between different graph representations in the HelloWorld-Example – is described as a relation between the model prior to the transformation (e.g., a graph without a super class for its elements) and the model after the transformation (e.g., a graph with a common super class for nodes and edges). In this section, the relations specifying the different transformations are described. To that end, a pure specification variant using only the previously introduced constructors is used, as well as a more pattern-oriented style, using add/delete patterns. The executable implementations of these transformations can be found in [5].

3.1 Declarative Relational Style

The constructors introduced in the previous section allow to break down and reconstruct a model from basic elements and relations. These elementary (de)construction predicates are sufficient to specify complex transformations. The `hello9` example in Subsection A.3 – showing the model transformation reversing the direction of the edges, by applying a recursive `reverseSrc` and `reverseTrg` relation – is a simple demonstration of the construction of complex transformations from these basic predicates. The transformation `hello9` is a relation linking the set consisting of pre-model `Pre` to the set consisting of post-model `Post`. The same predicate `graph1` is used in turn (lines 2, 6) to de- and reconstruct the models. Similarly, symmetric constructions are used to de-/re-construct the `src` and `trg` associations (lines 3, 5). Also the subtransformations `reverseSrc` and `reverseTrg` (lines 7, 11) use the same symmetric construction principle.

3.2 Pattern-Based Language Extension

The declarative relational style relies only on elementary predicates and therefore supports formal verification of properties of transformations in a straightforward way (see, e.g., [4]). However, on the downside, this low-level relation style lacks compactness of specification. Here, the introduction of an

additional language layer in terms of the lower-layer pure relational style can substantially improve the readability of specifications. To that end, basically, two specification patterns are introduced:

- a *deletion* pattern, in the form of `-class(+ClassName,?Entity,?Attr1,...,?AttrN)` and `-assoc(+AssocName,?Src,?Trg)` to remove an object `Entity` from class `ClassName` with attributes `Attr1,...,AttrN`, or a relation from association `AssocName` between `Src` and `Trg`;
- furthermore, an *introduction* pattern, in the form of `+class(+ClassName,?Entity,?Attr1,...,?AttrN)` and `+assoc(+AssocName,?Src,?Trg)` to add an object or a relation.

These patterns can be easily expressed in the declarative relational style. For example, the deletion of an element can be expressed as

```

1  ClassName(PreClass,PreSet), union([PreClass],RestClasses,PreClasses),
2  ClassName(Object,Element,Arg1,...,ArgN), union([Object],PostSet,PreSet),
3  ClassName(PostClass,PostSet), union([PostClass],RestClasses,PostClasses)

```

with `PreClasses` and `PostClasses` denoting to the classes before and after the application of the pattern. The patterns for the creation of objects and the deletion as well as the creation of relations are defined correspondingly. Subsection B.1 shows the complete formalization as used in the translation described as the end of this section.

These patterns are combined in the form of compositional and potentially recursive rules, which simply form lists of patterns. Using these mechanisms, the core functionality of `hello9` example from above can be rephrased as the rules provided in Subsection A.3:⁵

```

1  reverse([
2    -assoc(src,Edge,Node),
3    rule(reverse),
4    +assoc(trg,Edge,Node)]).
5  reverse([
6    -assoc(trg,Edge,Node)
7    rule(reverse),
8    +assoc(src,Edge,Node)]).
9  reverse([]).

```

plus `-` to provide the same headname for the rules – the pseudo rule

```

1  hello9([rule(reverse)]).

```

As in the case of the relation declarative style, rules are applied in the order of listing. Rules are only applied once – repeated rule application is achieved by the recursive application via `apply` in the rule body as shown, e.g., in the `reverse` rule of `hello9`.

To map the pattern-based language layer down to the declarative relational layer, an execution engine for the add-/delete patterns is provided in form of an interpreter in Prolog, using the core language, as described in Subsection B.2. Since the pattern-based language needs a possibility to support the execution of Prolog rules (e.g., to perform calculations or call other Prolog routines), an additional extension via the `exec` pattern is provided, taking a Prolog statement as its argument. This pattern is, e.g., used in the definition of the `hello7` example in Subsection A.2.

4 Conclusion and Outlook

⁵For ease of reading some Prolog required quotation marks have been dropped.

The PETE transformation framework – provided as an Eclipse PlugIn [3] – 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.

Some specific advantages visible in the case study concern the direct expressiveness of the transformation language. E.g., `hello2` and `hello3` show that a model construction can be interpreted as a model deconstruction; reversing the order of the constructors ensures an efficient execution.

The declarative nature is demonstrated in `hello7`, also including the application of a high-order operator (`findAll`), where characterization of a triplet is directly applied in the `isTriplet` predicate.

Finally, by using a simple pattern-based language extension directly expressible in the core language, the conciseness of the language can be substantially improved. It is worthwhile to note that a pattern-based approach may not always lead to a more concise description of a transformation, as shown in Subsection A.2.

The HelloWorld case shows that a relation, declarative, rule-based system like PETE is a suitable tool study properties of transformation languages, either by offering built-in properties like bi-directionality, or by allowing to quickly prototype more specialized formalisms as language layers on top of the core layer. However, this generality comes at a price: An important drawback of the current approach concerns the representation of sets as sorted lists, leading to a less efficient execution.

References

- [1] Steffen Mazanek (2011): *Hello World! An Instructive Case for the Transformation Tool Contest*. In Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors: *TTC, EPTCS*.
- [2] Bernhard Schätz (2008): *Formalization and Rule-Based Transformation of EMF Ecore-Based Models*. In Eric Van Wyk Dragan Gasevic, Ralf Laemmel, editor: *Software Language Engineering*, LNCS, Springer.
- [3] Bernhard Schätz (2009): *Prolog EMF Transformation Eclipse-PlugIn*. Available at www4.in.tum.de/~schaetz/PETE.
- [4] Bernhard Schätz (2010): *Verification of Model Transformations*. ECEASST 29.
- [5] Bernhard Schätz (2011): *SHARE demo related to the paper “Saying Hello World with PETE - A Solution to the TTC 2011 Instructive Case”*. Available at http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu-11_TTC11_PETE.vdi.
- [6] Dave Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2007): *EMF: Eclipse Modeling Framework*. Addison Wesley Professional. Second Edition.

A Code Listings

In Section 3 the two styles – the pure *declarative relational* and the *add/delete pattern-oriented* rule-based specification format - have been introduced. In the following, the solutions to the different tasks are presented in the corresponding styles.⁶

A.1 Hello World

The `hello1` relation create a single `Greeting` element, forms a `Greeting` class with one element, and wraps this in a `helloworld`-package.

⁶For ease of reading some quotation marks have been dropped for uppercase (de-)Constructors, which are required to distinguish them from variables defined in Problog by uppercase names. The complete code is provided in [5].

```

1 hello1([Pre],[Post]) :-
2   Greeting(Hello,Id,'Hello World'),
3   union({Hello},{},Greetings),Greeting(Greeting,Greetings),
4   union({Greeting},{},Classes), helloworld(Post,[],Classes,[]).

```

Correspondingly, the pattern-based style simply introduces a corresponding `Greeting` element.

```

1 hello1([
2   +class('Greeting',Hello,'Hello World')
3 ]).

```

The `hello2` relation creates `Greeting`, `GreetingMessage`, and `Person` elements, forms corresponding `Greeting`, `GreetingMessage`, and `Person` classes with one element, creates the corresponding `greetingMessage` and `person` relations between as well as the `greetingMessage` and `person` corresponding association set with one relations, and wraps this in a `helloworldext`-package.

```

1 hello2([Pre],[Post]) :-
2   Greeting(Greet,GElement), union({Greet},{},Greetings), Greeting(Greeting,Greetings),
3   GreetingMessage(Hello,MEElement,'Hello World'), union({Hello},{},Messages), GreetingMessage(Message,Messages),
4   Person(Participant,PEElement,'TTC Participant'), union({Participant},{},Persons),Person(Person,Persons),
5   greetingMessage(GreetMessage,GEElement,MEElement), union({GreetMessage},{},GreetingMessages), greetingMessage(
6     GreetingMessage,GreetingMessages),
7   person(GreetPerson,GEElement,PEElement), union({GreetPerson},{},GreetingPersons), person(GreetingPerson,
8     GreetingPersons),
9   union({Greeting,Message,Person},{},Classes), union({GreetingMessage,GreetingPerson},{},Assocs),
10  helloworldext(Post,[],Classes,Assocs).

```

The pattern-based style simply introduces the corresponding elements as well as their relations.

```

1 hello2([
2   +class('Greeting',Greeting),
3   +class('GreetingMessage',Message,'Hello World'),
4   +class('Person',Participant,'TTC Participant'),
5   +assoc(greetingMessage,Greeting,Message),
6   +assoc(person,Greeting,Participant)
7 ]).

```

`hello3` provides a model to text transformation, taking apart the model by simply using the same operations as above in the reversed order, and creates a corresponding `StringResult` element wrapped in a `result` package.

```

1 hello3([Pre],[Post]) :-
2   helloworldext(Pre,[],Classes,Assocs),
3   Greeting(Greetings,[Greeting]), GreetingMessage(Messages,[Message]), Person(Persons,[Person]),
4   union([Greetings,Messages,Persons],RestClasses,Classes),
5   Greeting(Greeting,GEElement), 'GreetingMessage'(Message,MEElement,Text), Person(Person,PEElement,Name),
6   greetingMessage(GreetingMessages,[GreetingMessage]), person(GreetingPersons,[GreetingPerson]),
7   union([GreetingMessages,GreetingPersons],RestAssocs,Assocs),
8   greetingMessage(GreetingMessage,GEElement,MEElement), person(GreetingPerson,GEElement,PEElement),
9   StringResult(Result,RElement,Text+Name), StringResult(Results,[Result]),
10  result(Post,[],[Results],[]).

```

The pattern-based style deletes the corresponding related `Greeting`, `GreetingMessage` and `Person` elements and introduces the `StringResult` element.

```

1 hello3([
2   -class('Greeting',Greeting),
3   -assoc(greetingMessage,Greeting,Message),
4   -class('GreetingMessage',Message,Text),

```

```

5  -assoc(person,Greeting,Person),
6  -class('Person',Person,Name),
7  +class('StringResult',Result,Text+Name)
8  ]).

```

A.2 Counting Matches

In `hello4`, after selecting the `Nodes` class from the `graph1` package, the Prolog built-in predicate `length` is used to count the size of the class.

```

1  hello4([Pre],[Post]):-
2      graph1(Pre,_,_GraphClasses,_),
3      Node(NodeClass,Nodes), union([NodeClass],_GraphClasses), length(Nodes,NodeSize),
4      IntResult(CountResult,RElement,NodeSize), union([CountResult],[],Results),
5      IntResult(Result,Results), union([Result],[],ResultClasses),
6      result(Post,[],ResultClasses,[]).

```

In `hello5`, after selecting the `src` and `trg` relations from the `graph1` package, the predicate `count` is used to search for edges with the same start and end node (first case), or stop counting otherwise (second case).

```

1  hello5([Pre],[Post]):-
2      graph1(Pre,_,_Assocs),
3      src(SrcAssoc,Srcs), trg(TrgAssoc,Trgs), union({SrcAssoc,TrgAssoc},RestAssocs,Assocs),
4      count(Srcs,Trgs,EdgeNumber),
5      IntResult(CountResult,RElement,EdgeNumber), union({CountResult},{},Results),
6      IntResult(Result,Results), union({Result},{},ResultClasses),
7      result(Post,[],ResultClasses,[]).
8  count(Srcs,Trgs,Number):-
9      union([SrcRel],RestSrcs,Srcs), src(SrcRel,Edge,Node),
10     union([TrgRel],RestTrgs,Trgs), trg(TrgRel,Edge,Node),
11     count(RestSrcs,RestTrgs,RestNumber),
12     Number is RestNumber + 1.
13 count(Srcs,Trgs,Number):- Number is 0.

```

In `hello7`, after selecting the `src` and `trg` relations from the `graph1` package, the predicate `isTriplet` – defining a edge triplet of three nodes `Node1`, `Node2`, and `Node3` in a `Srcs` and `Trgs` association – is used to search for all matching trippels via the built-in Prolog predicate `findall` – returning all those instances – and counting the result set.

```

1  hello7([Pre],[Post]):-
2      graph1(Pre,_,_Classes,Assocs),
3      src(SrcAssoc,Srcs), trg(TrgAssoc,Trgs), union([SrcAssoc,TrgAssoc],RestAssocs,Assocs),
4      findall([Node1,Node2,Node3],isTriplet(Srcs,Node1,Node2,Node3,Trgs),ResultList), length(ResultList,Number),
5      IntResult(CountResult,RElement,Number), union({CountResult},{},Results),
6      IntResult(Result,Results), union({Result},{},ResultClasses),
7      result(Post,[],ResultClasses,[]).
8  isTriplet(Srcs,Node1,Node2,Node3,Trgs):-
9      src(Src1,Edge1,Node1), union({Src1},RestSrcs1,Srcs), trg(Trg1,Edge1,Node2), union({Trg1},RestTrgs1,Trgs),
10     src(Src2,Edge2,Node2), union({Src2},RestSrcs2,RestSrcs1), trg(Trg2,Edge2,Node3), union({Trg2},RestTrgs2,RestTrgs1),
11     src(Src3,Edge3,Node3), union({Src3},RestSrcs3,RestSrcs2), trg(Trg3,Edge3,Node1), union({Trg3},RestTrgs3,RestTrgs2).

```

The pattern-based approach to the solution demonstrates that in some cases – especially when the meta-models of the source and the target model differ – a pattern-like description of rules may have disadvantages. Since the pattern-based approach uses an in-place mechanism, the source-model must be

deconstructed and thus removed to be replaced by the target model. This additional part is taken over by the cleanUp rule.

```

1 hello7([
2   rule(count),
3   rule(cleanUp)
4 ]).
5 count([
6   -class(Edge,Edge1),-class(Edge,Edge2),-class(Edge,Edge3),
7   -assoc(src,Edge1,Node1),-assoc(trg,Edge1,Node2),
8   -assoc(src,Edge2,Node2),-assoc(trg,Edge2,Node3),
9   -assoc(src,Edge3,Node3),-assoc(trg,Edge3,Node1),
10  -assoc(edges,Graph,Edge1),-assoc(edges,Graph,Edge2),-assoc(edges,Graph,Edge3),
11  rule(count),
12  -class(IntResult,Result,OldValue),
13  exec(is(NewValue,OldValue+1)),
14  +class(IntResult,Result,NewValue)
15 ]).
16 count([
17   +class(IntResult,Result,0)
18 ]).
19 cleanUp([
20   -class(Node,Node,Name),
21   -assoc(nodes,Graph,Node),
22   rule(cleanUp)
23 ]).
24 cleanUp([
25   -class(Edge,Edge),
26   -assoc(src,Edge,Src),
27   -assoc(trg,Edge,Trg),
28   -assoc(edges,Graph,Edge),
29   rule(cleanUp)
30 ]).
31 cleanUp([-class(Graph,Graph)]).

```

A.3 Reverse Edges

hello9 shows a model transformation reversing the direction of the edges, by applying a recursive reverseSrc and reverseTrg relation.

```

1 hello9([Pre],[Post]) :-
2   graph1(Pre,[],Classes,OldAssocs),
3   src(OldSrcAssoc,OldSrcs), trg(OldTrgAssoc,OldTrgs),union ({OldSrcAssoc,OldTrgAssoc},RestAssocs,OldAssocs),
4   reverseSrc(OldSrcs,NewSrcs), reverseTrg(OldTrgs,NewTrgs),
5   src(NewSrcAssoc,NewSrcs), trg(NewTrgAssoc,NewTrgs),union ({NewSrcAssoc,NewTrgAssoc},RestAssocs,NewAssocs),
6   graph1(Post,[],Classes,NewAssocs).
7 reverseSrc(Srcs,Trgs) :-
8   src(Src,Edge,Node), union ({Src},RestSrcs,Srcs), trg(Trg,Edge,Node), union ({Trg},RestTrgs,Trgs),
9   reverseSrc(RestSrcs,RestTrgs).
10 reverseSrc([],[]).
11 reverseTrg(Trgs,Srcs) :-
12   trg(Trg,Edge,Node), union ({Trg},RestTrgs,Trgs), src(Src,Edge,Node), union ({Src},RestSrcs,Srcs),
13   reverseTrg(RestTrgs,RestSrcs).
14 reverseTrg([],[]).

```

Correspondingly, the pattern-based formalization uses corresponding rules, to remove a relation and add its swapped relation.

```

1 hello9([rule(reverse)]).
2 reverse([
3   -assoc(src,Edge,Node),
4   rule(reverse),
5   +assoc(trg,Edge,Node)]).
6 reverse([
7   -assoc(trg,Edge,Node)
8   rule(reverse),
9   +assoc(src,Edge,Node)]).
10 reverse([]).

```

A.4 Simple Migration

hello10 shows the migration by introduction of a joint superclass of nodes and egdes.

```

1 hello10([Pre],[Post]) :-
2   graph1(Pre,[],PreClasses,PreAssocs),
3   Edge(PreEdgeClass,PreEdges), union({PreEdgeClass},RestClasses,PreClasses),
4   edges(EdgesAssoc,GraphEdges), nodes(NodesAssoc,GraphNode), union({EdgesAssoc,NodesAssoc},RestAssocs,
5     PreAssocs),
6   migrateEdges(PreEdges,PostEdges),
7   migrateGCs(GraphEdges,GraphNode,GraphGCs),
8   gcs(GCsAssoc,GraphGCs), union({GCsAssoc},RestAssocs,PostAssocs),
9   Edge(PostEdgeClass,PostEdges), union({PostEdgeClass},RestClasses,PostClasses),
10  graph2(Post,[],PostClasses,PostAssocs).
11 migrateEdges(PreEdges,PostEdges) :-
12   union({PreEdge},PreRest,PreEdges), Edge(PreEdge,EdgeElem),
13   migrateEdges(PreRest,PostRest),
14   Edge(PostEdge,EdgeElem, ' '), union([PostEdge],PostRest,PostEdges).
15 migrateEdges([],[]).
16 migrateGCs(Edges,GraphNode,GraphGCs) :-
17   union([EdgesRel],RestEdges,Edges), edges(EdgesRel,Graph,Edge),
18   migrateGCs(RestEdges,GraphNode,RestGCs),
19   gcs(GCRel,Graph,Edge), union({GCRel},RestGCs,GraphGCs).
20 migrateGCs([],Nodes,GraphGCs) :-
21   union({NodesRel},RestNodes,Nodes), nodes(NodesRel,Graph,Node),
22   migrateGCs([],RestNodes,RestGCs),
23   gcs(GCRel,Graph,Node), union({GCRel},RestGCs,GraphGCs).

```

Similary, the pattern-based formalization uses corresponding rules to delete and introduce an Edge in the source and target model, correspondingly, as well as delete the edges as well as nodes relation and introduce the gcs relation.

```

1 hello10([rule(migrateEdges),rule(migrateGCs)]).
2 migrateEdges([
3   -class('Edge',Edge),
4   rule(migrateEdges),
5   +class('Edge',Edge,'')]).
6 migrateEdges([]).
7 migrateNodes([
8   -class('Node',Node,Name),
9   rule(migrateNodes),

```

```

10  +class('Node',Node,Name)].
11  migrateNodes([]).
12  migrateGCs([
13    -assoc(edges,Graph,Edge)
14    rule(migrateGCs),
15    +assoc(gcs,Graph,Edge)].
16  migrateGCs([
17    -assoc(nodes,Graph,Node)
18    rule(migrateGCs),
19    +assoc(gcs,Graph,Node)].
20  migrateGCs([]).

```

hello11 shows the deletion of a node and its adjacent edges, using the functors directly without the use of the corresponding constructor predicates..

```

1  hello11([Pre],[Post]):-
2    graph1(Pre,[],OldClasses,OldAssocs),
3    union({'Node'(OldNode),'Edge'(OldEdge)},RestClasses,OldClasses),
4    union({'src'(OldSrcs),trg'(OldTrgs),nodes'(OldNodes),edges'(OldEdges)},RestAssocs,OldAssocs),
5    deleteNode(OldNode,OldEdge,OldSrcs,OldTrgs,OldNodes,OldEdges,NewEdges,NewNodes,NewTrgs,NewSrcs,NewEdge,
6      NewNode),
7    union({'src'(NewSrcs),trg'(NewTrgs),nodes'(NewNodes),edges'(NewEdges)},RestAssocs,NewAssocs),
8    union({'Node'(NewNode),'Edge'(NewEdge)},RestClasses,NewClasses),
9    graph1(Post,[],NewClasses,NewAssocs).
10 deleteNode(OldNode,OldEdge,OldSrcs,OldTrgs,OldNodes,OldEdges,
11   NewEdges,NewNodes,NewTrgs,NewSrcs,NewEdge,NewNode):-
12   union({'Node'(Node,n1)},NewNode,OldNode), union({'nodes'(Graph,Node)},NewNodes,OldNodes),
13   deleteEdges(OldEdge,OldSrcs,OldTrgs,OldEdges,Node,NewEdges,NewTrgs,NewSrcs,NewEdge).
14 deleteNode(Node,Edge,Srcs,Nodes,Edges,Edges,Nodes,Trgs,Srcs,Edge,Node).
15 deleteEdges(OldEdge,OldSrcs,OldTrgs,OldEdges,Node,NewEdges,NewTrgs,NewSrcs,NewEdge):-
16   union({'src'(TheEdge,Node)},Srcs,OldSrcs), union({'trg'(TheEdge,Other)},Trgs,OldTrgs),
17   union({'edges'(Graph,TheEdge)},Edges,OldEdges), union({'Edge'(TheEdge)},Edge,OldEdge),
18   deleteEdges(Edge,Srcs,Trgs,Edges,Node,NewEdges,NewTrgs,NewSrcs,NewEdge).
19 deleteEdges(OldEdge,OldSrcs,OldTrgs,OldEdges,Node,NewEdges,NewTrgs,NewSrcs,NewEdge):-
20   union({'trg'(TheEdge,Node)},Trgs,OldTrgs), union({'src'(TheEdge,Other)},Srcs,OldSrcs),
21   union({'edges'(Graph,TheEdge)},Edges,OldEdges), union({'Edge'(TheEdge)},Edge,OldEdge),
22   deleteEdges(Edge,Srcs,Trgs,Edges,Node,NewEdges,NewTrgs,NewSrcs,NewEdge).
23 deleteEdges(Edge,Srcs,Trgs,Edges,Node,Edges,Trgs,Srcs,Edge).

```

B Language Layering Mechanism

In the following, the language layering mechanisms – the pattern formalization and the interpreter – are described in more detail.

B.1 Pattern Rule Formalization

To effectively translate the patterns, the class and association names have to be applied to selected the corresponding constructors. The following listing demonstrates the use of the Prolog = . . and *call* predicates in the formalization of the *delete class* pattern:

```

1  -class((PreClasses,Assocs),[ClassName|Arguments],[PostClasses,Assocs]):-
2    PreClassTerm =..[ClassName,PreClass,PreSet],call(PreClassTerm),union([PreClass],RestClasses,PreClasses),
3    ObjectTerm =..[ClassName,Object|Arguments],call(ObjectTerm),union([Object],PostSet,PreSet),
4    PostClassTerm =..[ClassName,PostClass,PostSet], call(PostClassTerm),union([PostClass],RestClasses,PostClasses).

```

For this translation, sub-packages are not supported by the interpreter.

B.2 Prolog Interpreter for the Pattern Language

To map the pattern-based language layer down to the declarative relational layer, as mentioned in Section 1, an execution engine for the add-/delete patterns is provided by adding an interpreter in Prolog, using the core language. The following listing shows the complete code for the interpreter. It makes use of the `=..` term (de)constructor provided by Prolog, as well as the *call*-predicate, executing the constructed terms.

```
1 apply(Pre,Rule,Post) :-  
2     RuleTerm =.. [Rule,Constraints], call(RuleTerm), execute(Pre,Constraints,Post).  
3 execute(Pre,[Constraint|Constraints],Post) :-  
4     Constraint =.. [Type|Arguments], Declaration =.. [Type,Pre,Arguments,Intermediate],  
5     call(Declaration), execute(Intermediate,Constraints,Post).  
6 execute(Model,[],Model).  
7 rule(Pre,[Rule],Post) :-  
8     apply(Pre,Rule,Post).
```