A Framework for Executing Cross-Model Transformations Based on Pluggable Metamodels

Sven De Labey *Geert Delanote*
Koen Vanderkimpen *Eric Steegmans*

Report CW 489, May 2007

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A Framework for Executing Cross-Model Transformations Based on Pluggable Metamodels

Sven De Labey *Geert Delanote*
Koen Vanderkimpen *Eric Steegmans*

Report CW489, May 2007

Department of Computer Science, K.U.Leuven

Abstract

The design of complex software systems requires developers to use a variety of modeling languages in order to model various system aspects. The heterogeneity of these modeling languages gives rise to new challenges. Design decisions must be communicated across heterogeneous models, thus creating a need for *cross-model communication*. Furthermore, models must be transformable between different modeling languages, thus creating a need for *cross-model transformations*. By supporting only a single modeling language and by providing limited interoperability, however, the majority of today's modeling tools cannot provide cross-model communication nor transformation, as such jeopardizing the consistency of the design as a whole.

In this paper, we present the design of a *transformation framework*, Pluto, which supports *cross-model transformations* based on *pluggable metamodels*. We discuss how Pluto eases the realization of concrete metamodels by offering abstract modeling constructs, and we show how it is able to execute transformations between concrete instances of such metamodels.

Keywords : Model Transformations, Metamodeling, Framework

1 Introduction

In a previous document, we have presented the state of the art of current languages and techniques used for specifying and executing model transformations. In that overview, we focused on providing a *taxonomy* of transformation techniques, including approaches such as imperative and declarative transformation languages, XML-based and graph-based transformation approaches, and even tools for specifying transformation definitions using a graphical notation.

An important drawback of the approaches outlined in that document, was that most of them relied on the *hardwired metamodel* of a *single modeling language*. Most UML modeling tools that support transformations, for instance, rely on a hardwired, vendor-specific UML metamodel. By hardwiring their metamodels, such tools disable transformations to *other* modeling languages, and they must be revised each time the UML metamodel is *altered* or *extended*. As ad hoc metamodels are typically entangled with the code of the transformation tool itself, such revisions and extensions are everything but trivial. Additionally, by relying on vendor-specific metamodels, transformation tools obstruct metamodel reuse in other modeling tools, as those tools also rely on ad hoc, vendor-specific (and hence incompatible) metamodels.

Being convinced that vendor lock-in and lack of support for cross-model transformations are serious limitations of today's modeling tools, we have designed and implemented a transformation tool that relies on *interchangeable metamodels*. Our tool has two important advantages relative to existing approaches. First, the tool is extensible because it allows to plug in metamodels that did not even exist during the implementation of the transformation tool. This is a significant advantage compared to traditional transformation tools, in which the metamodel is hardwired and entangled during the development of the tool. Second, our tool provides transformation implementors with a higher level of abstraction by introducing a number of transformation rules that are enforced automatically by our transformation tool.

This document is structured as follows. Section 2 gives an overview of the design goals that were pointed out during the analysis of the project. Section 3 outlines the Pluto framework. Sections 4–6 discuss the Pluto classes for metamodeling and show how these framework classes are reused during the implementation of a new metamodel. In this text, we use the Entity-Relation and the Relational Database metamodels as examples of concrete metamodels. Sections 7–10 focus on model transformations and show how cross-model transformations can be realized using the concrete metamodels (ER and RDB) of section 6. Finally, section 11 presents related work, section 12 presents some future work and section 13 concludes.

2 Design Goals

The design goals that were defined for this project, range from straightforward functional requirements (e.g. transformations) over non-functional requirements (e.g. reusability and code comprehension) to important contributions such as cross-model transformations and metamodel extensibility. The following list formally states our objectives:

- **Model Transformations.** The fundamental design goal is that our tool must be able to execute transformations between models based on a shared metamodel. This is the functionality that most of today's transformation tools support [5].
- **Cross-Model Transformations.** Next to transformations between models that share a common metamodel, the tool must support transformations between different modeling languages [25, 8, 10]. This implies that the tool must be able to work with *separate* source and target metamodels. Unfortunately, tools supporting this characteristic are not wide-spread because their underlying metamodel is entangled with the code of the transformation engine. Most commercial UML modeling tools, for example, rely on hardwired, non-standard UML metamodels. Our transformation tool must avoid such dependencies and must therefore rely on a *general, model-independent specification*, which serves as a contract stating the minimal functionality that metamodels must exert in order to be readable and usable by the transformation tool.
- **Reusable.** Our transformation tool relies on interchangeable metamodels and these metamodels must be *reusable* in other domains. Indeed, metamodels have a wide-spread applicability in, for instance, design tools or code generators, and we want to make sure that they can be reused in those tools without modification. From the viewpoint of our project, this goal implies that we must avoid polluting the metamodels with any references to transformation classes. In a way, this is similar to the Inversion of Control (IoC) principle, a popular dependency inversion mechanism in today's middleware applications: the metamodel itself does not know how it is transformed, whereas *an external force* (the transformation tool) works on that metamodel so as to fulfill the transformation. As will be shown in the remainder of this text, the implementation of this mechanism was validly identified as one of the higher risks during earlier analyses of our project.
- **Extensibility.** Developers must be able to feed *newly defined metamodels* to the transformation tool, as such increasing its applicability. Our transformation tool running on a UML metamodel must be able to accept, for instance, an Entity Relation metamodel or a Relational Database metamodel and then allow developers to design database models in those modeling languages. Combined with the desired ability to

execute cross-model transformations (as discussed above), this means that the tool must be able to transform between UML, ER, and RDB.

- **Modularity.** Being closely related to extensibility, *modularity* requires that decisions concerning the transformation model are kept isolated from decisions concerning the *source* or *target metamodels*. Thus, a certain amount of *obliviousness* is needed in both directions: on one hand, the metamodels should be entirely independent of the transformation tool (see: *reusability*); on the other hand, a transformation tool should not depend on the technical details of one metamodel (see: *cross-model transformations*). This observation justifies the introduction of a *rule set* on which the transformation tool relies when a metamodel is fed to it. Such *model-independent contracts* allow us to evolve the transformation tool independently from the metamodels, and vice versa, thus achieving a higher degree of modularity.

- **Code comprehension.** Stating general objectives such as *reusability* and *extensibility* is sensible only if our tool is actually reused and extended. This is only feasibile if our prototype is sufficiently documented and if its API is rigorously specified. While slightly increasing the design time, rigorous specifications dramatically decrease maintenance costs as well as the probability to introduce bugs in subsequent releases of the transformation tool. Moreover, by increasing the overall comprehensibility of our code, we decrease both the time and the cost for adding extra metamodels by third parties, which is important because such extensions eventually comprise the value-adding factor of our transformation tool.

Long term objectives. By realizing the need for cross-model transformations, combined with the requirement to support the addition of new metamodels, we pave the road towards *network effects*. Indeed, each newly introduced metamodel will eventually allow us to transform instances of that new metamodel into models of the already added metamodels, thus firmly increasing the applicability of our transformation tool. Another advantage of having an extensible, model-independent transformation tool at our disposal, is that we can define our own modeling language, assemble its definition into a new metamodel, and feed it to our transformation tool. The tool will then be able to transform models defined using our own language into models of existing languages, without forcing developers to revise or extend the source code of the transformation framework.

3 Architecture Overview

Before elaborating on the details of our project, we provide the reader with a *conceptual view* on the architecture of the transformation tool. All technical details and implementation issues are deferred to further sections of this text. In this section, we show how our tool is

based on the Pluto Metamodel (1) for defining and *implementing metamodels*, as shown in section 3.1, and (2) for *implementing transformations*, as discussed in section 3.2.

3.1 How Pluto supports the Implementation of Metamodels

Current transformation tools typically operate on hardwired metamodels, thus obstructing cross-model transformations. We solve this problem by no longer relying on *hardwired* metamodels and make the metamodel *interchangeable* instead. As such, we allow modelers to insert their own metamodels without forcing them to write their own transformation tool from scratch. This section outlines the Pluto framework and shows how the definition of new metamodels is simplified by allowing metamodel implementors to reuse *common metamodel constructions* offered by the Pluto framework.

Reusable metamodel concepts. Metamodels typically share a lot of common functionality. For example, they typically have to manage dependency relations, such as *parent-child* relations, leading to a tree-like representation of the metamodel. A concrete example of such a dependency is found in the Entity-Relation metamodel, where a *strong entity* (a child) belongs to exactly one *ER model* (the parent). This child can in turn be the parent of other children, as such leading to a dependency tree. Next to tree structures, a lot of other common functionality is found in metamodels of modeling languages. Therefore, we have designed and implemented the Pluto *framework*, which provides *reusable constructions* for building concrete metamodels. Technically, these concrete metamodels are *subclasses* of classes offered by the Pluto framework.

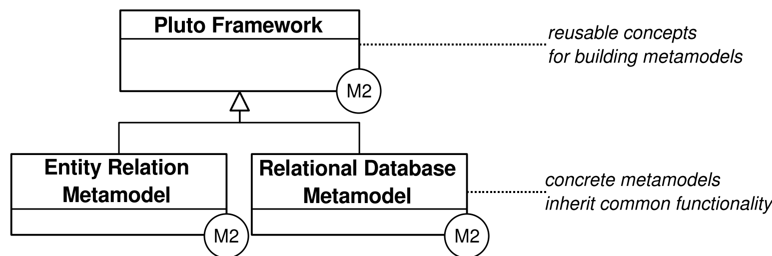

Figure 1: Pluto constitutes a general layer above concrete metamodels

Figure 1 shows how the Entity-Relation metamodel and the Relational Database Model are defined as specific extensions of the Pluto framework. Specific classes for the ER model (e.g. **Entity**) and specific classes for RDB models (e.g. **Relation**) are defined at the level of their concrete metamodels; at the same time, these concrete metamodels inherit all common functionality (e.g. dependency management) from the Pluto framework.

To avoid any confusion in the remainder of this text, it is also paramount to define what Pluto *is not*:

- *Pluto is not a metamodel itself.* Pluto provides reusable constructs for building metamodels, but it is not a metamodel itself; it lacks concrete concepts and it does not describe a concrete modeling language. The ER and the RDB metamodel, on the other hand, are full-fledged metamodels because they describe the ER and RDB models, respectively.
- *Pluto is not a metametamodel.* Although Pluto provides abstractions to be reused by concrete metamodels, it is not a metamodel of those concrete metamodels. Therefore, Pluto is not a metametamodel, which explains why it is found at the M2 level of the Meta Object Facility, rather than at M3, the level of the metametamodel (see also figure 1).

More Information. Detailed information about how Pluto supports metamodel *definitions* is given in section 4. A complete *API reference* is given in section 5. Also, *concrete examples* of metamodels based on general Pluto classes are shown in section 6.

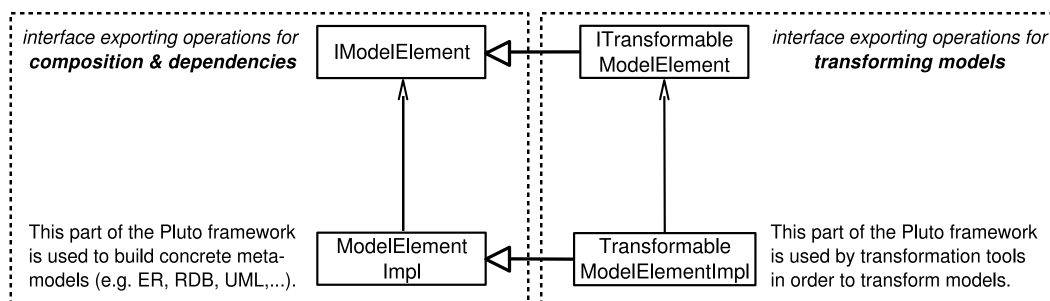

Figure 2: Transformable behaviour is added using inheritance

3.2 How Pluto supports the Implementation of Transformations

An important objective of our project is to leverage obliviousness between the transformation tool and the metamodels on which it operates. One major goal related to this general objective is to make metamodels *fully agnostic* about the transformation tool [19]. In other words, we are not allowed to insert transformation logic into the metamodel itself. The solution that we propose is to enrich model elements with transformation logic by means of *inheritance*. We introduce two interfaces for metamodel constructs: (1) a *general interface* containing operations for managing dependencies in metamodels (shown as **IModelElement** in figure 2), and (2) a *subinterface* which adds operations for transforming these

model elements (shown as **ITransformableModelElement** in figure 2). A detailed version of this inheritance structure is shown in figure 3. This figure also shows how the ER metamodel can be made transformable. The concept of an ER entity, for instance, can be made transformable by introducing a class that implements **ITransformableEntity**. Indeed, this interface inherits behaviour from **IEntity** (thus making it an entity) as well as from **ITransformableNonRootComposedModelElement** (thus making it transformable).

Developers of a modeling tool that does not support model transformations only have to implement the general interface (or they can extend a default implementation, as shown in figure 2). If needed, these *untransformable* metamodels can be made transformable by *subclassing* those classes of the metamodel that represent modeling concepts. These newly introduced subclasses implement the *subinterface* (e.g. **ITransformableModelElement**), as such attaching transformable behaviour to that model element. Doing so, the original metamodel, which only implements the *general interface*, remains unaware of any transformation code; the transformation tool, on the other hand, can effectively operate on model elements through the operations offered by the subinterface, which is implemented by the subclass (e.g. **ITransformableModelElementImpl** in figure 2).

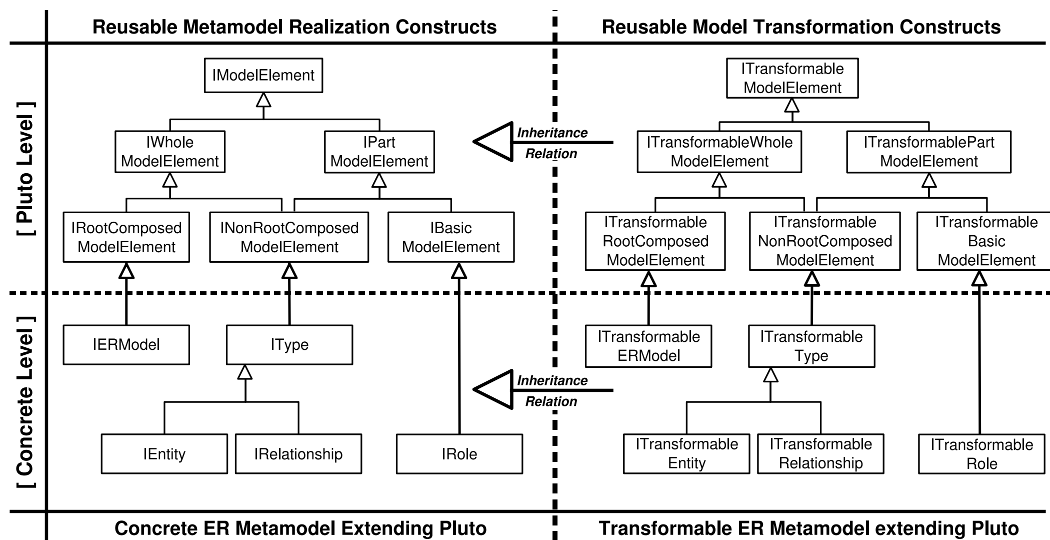

Figure 3: Overview of the Pluto framework for building metamodels (left part) and for attaching transformable behaviour to metamodel constructs (right part).

Obliviousness. It is very important to note that we have intentionally kept the flow in Figure 3 *unidirectional* in order to achieve our design goals related to metamodel obliviousness. Thus, reusable constructs for building metamodels will *never* inherit from concepts that

were introduced to make metamodels transformable. Such design decisions would decrease the reusability of concrete metamodels in applications where model transformations are not required.

Reusable transformation algorithm. Next to offering concepts for decorating metamodels with the ability to transform model elements, Pluto also provides a general algorithm for *executing* model transformations. Such transformation algorithms have to abide by a number of *transformation rules*. These rules often refer to general dependencies between model elements that were already supported by Pluto (see section 3.1), thus enabling us to implement a general transformation algorithm in Pluto without a loss of generality. Obviously, the more rules are satisfied at an abstract, reusable level such as Pluto the lesser the probability that a developer defines or implements an erroneous model transformation.

3.3 Structure of this Document

This section provides a quick overview of the remainder of this document. This text comprises two parts, as shown in figure 4. The *first part* contains detailed information about the conceptual design and the technical specification of Pluto for *metamodeling*. Section 4 provides a detailed discussion about the design decisions that underly the Pluto framework. Section 5 describes the API specification and points out some important implementation issues. Section 6 then shows how concrete metamodels can be implemented by extending Pluto classes. The *second part* of this text deals with the design of the transformation tool. Section 7 first extends the Pluto framework with classes for transforming metamodels. The API of these classes is shown in Section 8. The realization of concrete metamodels that extend these classes is discussed in Section 9. Finally, Section 10 shows how Pluto provides a reusable model transformation strategy and explains how this strategy can be tailored to the needs of concrete metamodels.

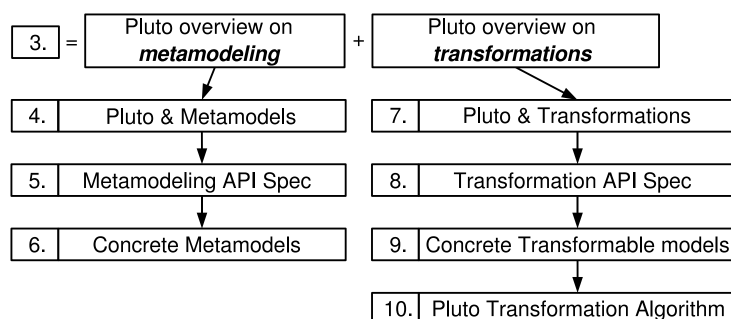

Figure 4: The structure of this document

4 Design of the Pluto Framework for Metamodeling

Before discussing the API specification of the Pluto framework, we give a conceptual introduction to its internal structure by pointing out a number of design decisions. First, we describe the pattern that most influences the overall architecture: Composite (section 4.1). Then, we introduce two dependency relations that are used extensively in Pluto and its concrete metamodels: *parent-child* and *dependee-dependant* (section 4.2). Finally, we describe how our tool ensures model consistency using *valid/invalid states* (section 4.3).

4.1 Composing Model Elements with the Composite Pattern

As already noted before, metamodels typically exhibit a tree-like structure where certain metamodel constructs are *composed* out of other metamodel constructs. The implementation of a metamodel therefore requires developers to design *whole/part* relationships. Support for such relations is integrated in Pluto using the Composite pattern [9].

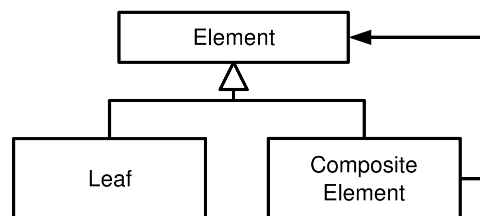

Figure 5: The basic Composite pattern

4.1.1 The Basic Composite Pattern

The basic structure of the Composite pattern is shown in figure 5. This figure shows how elements can be either *leaves* or *composite elements*. A **CompositeElement** can have an indefinite number of children, that conform to **Element**, the common supertype of leaves and composite elements. Thus, children of a **CompositeElement** may again be composite elements, giving rise to hierarchical data structures. The operations that can be invoked on members of such a tree structure comprise those operations exported by **Element**, the common ancestor of leaves and internal nodes.

4.1.2 Application of the Composite Pattern in Pluto

The implementation of the Composite pattern in Pluto slightly differs from the Composite pattern as described above in that our implementation introduces *bidirectional* associations between containers (*wholes*) and their contained model elements (*parts*). We have also introduced a *middle layer* in the Composite pattern, as shown in figure 6. This middle layer represents the *wholes* and *parts* of the basic Composite pattern, and is further specialized by three interfaces:

- **Root**. The root of a metamodel tree is modeled by **IRootComposedModelElement**. This is a container for other elements and, therefore, it extends the **IWholeModelElement** interface. On the other side, being the *root* of a tree, it cannot be contained by another model element, so it may not extend the **IPartModelElement** interface.
- **Leaves.** Leaves are represented by the **IBasicModelElement** interface. They are always contained in other model elements, which are either internal nodes or the root node. Also, leaves do not contain model elements themselves. Therefore, they only extend the **IPartModelElement** interface.
- **Internal nodes.** These nodes are represented by **INonRootModelElement**. Internal nodes are the actual *composite elements* of the tree. On one hand, they are containers for other elements, so they extend the **IWholeElement** interface, similar to the root node of the tree. On the other hand, they also exhibit the behaviour of a leaf, because internal nodes are contained in another model element (which may in turn be an internal node or the root node). Therefore, these nodes also extend the **IPartModelElement**.

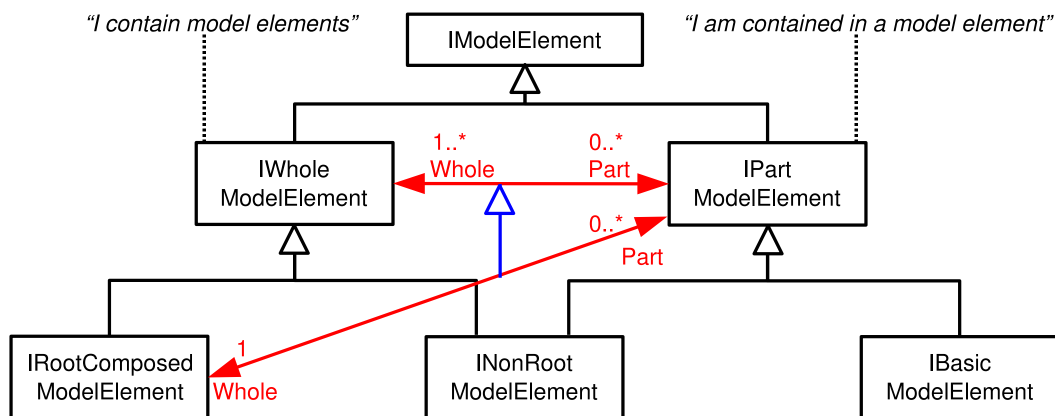

Figure 6: Extended Composite pattern used in Pluto

Semantics of the Whole-Part Relation. The Whole-Part relation is defined such that a model element can contain indefinitely many parts (**0..***), whereas each part is contained

in at least one whole. This typically means that a container survives the deletion of all of its parts, whereas the deletion of a container is cascaded to every contained part. Note, also, that the whole-part relation is *strengthened* between instances of **IRootComposed-ModelElement** and instances of **IPartModelElement**. This is done in order to avoid that a model element can have *multiple root nodes*. By enforcing this rule at the general level of **IPartModelElement**, we still allow the definition of models where both composed and basic elements are placed directly under the root node of the tree because **IBasicModelElement** and **INonRootModelElement** both conform to the **IPartModelElement** interface.

The next section elaborates on the dependencies between model elements in the context of this project. This elaboration causes the Whole-Part relation to be split into two different dependency relations, namely: *Parent-Child* and *Dependee-Dependant*.

4.2 Hierarchical Parent-Child and Dependee-Dependant Relations

Section 4.1 made clear that the Composite pattern is useful for modeling whole-part dependencies between model elements. This section specializes this view on dependencies into two different dependency relations between model elements: *unidirectional dependencies* and *interdependencies*. Both frequently occur in metamodels, so we have included support for managing such relations in our Pluto framework. The *parent-child* relation is used to specify *interdependencies*. This means that the parent is existentially dependent on its children, and vice versa. The dependee-dependant relation, on the other hand, is used to specify *unidirectional relationships*, meaning that a dependant is existentially dependent on its dependees, although the inverse is not true. These relations are discussed in section 4.2.1 (parent/child) and in section 4.2.2 (dependee/dependant). An example illustrating both dependency relationships is given in section 4.2.3. Finally, in section 4.2.4, we show how these specialized dependency relations integrate with our extended version of the Composite pattern.

4.2.1 Parent-Child Relations specify Model Interdependencies

The parent-child relationship is a hierarchical one-to-many relation: a parent can have many children, whereas a child only has one parent. The following list summarizes general characteristics about the parent-child relation.

- **One-to-Many.** A parent can have multiple children, but a child only has one parent. For example, the ER metamodel specifies that a relationship (parent) can have many participating roles (children), but a role can only be part of one relationship.

- **Interdependence.** All children depend on their parent, and the parent depends on its children (although this inverse dependency is typically weaker):
  - **Children depend on their parent.** The parent-child relation installs a cascading delete strategy, meaning that the children cannot survive the removal of a parent. In the ER metamodel, for instance, this means that relationship roles cannot survive the removal of the relationship to which they belong.
  - **Parent depends on children.** In parent-child relations, the parent also depends on its children. The validity of a relationship, for instance, is influenced by the existence of relationship roles; without roles, the parent would not be a valid relationship because there are no participants.
- **Transitive.** Given three model elements **A**, **B**, and **C**, such that **A** is a parent of **B** and **B** is a parent of **C**, then **A** is an indirect parent (or *ancestor*) of **C**.
- **Symmetric.** If a model element **A** is the parent of another model element **B**, then **B** is a child of **A**. This property requires the bidirectional association between parents and their children to be kept consistent at all times. By integrating dependency management at the level of Pluto however, these constraints are satisfied automatically.
- **Non-reflexive and Acyclic.** No model element can be a parent of itself. Additionally, a child cannot be the parent of one of its ancestors. Consequently, the data structure, induced by transitively applying the parent-child relation one or more times, is *acyclic*.
- **Tree structure.** Having the abovementioned properties, we can construct the non-reflexive, transitive closure of the parent-child relationship. This closure induces an acyclic tree structure on a set of model elements. The structure is a tree rather than a lattice due to the one-to-many property, which states that a child only has *one parent*. This simplifies the implementation of iterators that run over a set of related model elements.

4.2.2 Dependee-Dependant Relations specify Model Dependencies

The dependee-dependant relation is a hierarchical many-to-many relationship, meaning that a dependant relies on one or more dependees, which in turn have zero or more dependants. The properties of the dependee-dependant relation are summarized below:

- **Many-to-Many.** Other than parent-child interdependencies, where a child only has one parent, a dependant can have multiple dependees, thus leading to a many-to-many multiplicity for dependee-dependant relations. Currently, our ER metamodel

contains at most one dependee for each dependant, which would justify the specification of a one-to-many dependency relation. We have decided, however, to keep the many-to-many relation for extensibility reasons. Indeed, other metamodels besides the ER metamodel, might need to register multiple dependees for a dependant.

- **Unidirectional dependency relation.** Unlike the parent-child relation, where a parent depends on its children, the dependees of the dependee-dependant relation are *independent* of their dependants. In the ER metamodel, for instance, a weak entity is a dependee of its attributes (the dependants) and the attributes can be removed without making the weak entity inconsistent.
- **Non-reflexive, symmetric, transitive, acyclic.** The dependee-dependant relation has the same mathematical properties as the parent-child relation. By installing a many-to-many dependency relation, however, the non-reflexive, transitive closure of the dependee-dependant relation will induce an acyclic *lattice* structure on the depending model elements, rather than an acyclic *tree*.

4.2.3 Example

Figure 7 shows an example of how the parent-child and dependee-dependant relations are used in a part of the ER metamodel. A detailed discussion of this metamodel is presented in section 6.

Example of a Parent-Child Relation. In figure 7, **Entity** is a *parent* of **PrimaryKey** because, obviously, a primary key cannot exist without being attached to a strong entity. Conversely, a strong entity cannot exist without having at least one key; otherwise it would be a *weak* entity. Therefore, keys *influence* the validity of **Entity**, and the *interdependence* between both model elements forces the use of a parent-child relation. Similar interdependencies exist between a **Relation** and a **Role** representing a participant of that relation. Indeed, roles can only exist when they are contained in a relation, and a relation is only sensible when it has at least one role. These dependencies will be explained in section 6, where a discussion on concrete metamodels extending the Pluto framework is presented. At this point, is should be clear that by integrating the parent/child relation at the level of Chameleon, developers of concrete metamodels may reuse functionality for managing interdependencies between model elements without having to check for the complex constraints mentioned in section 4.2.1.

Example of a Dependee-Dependant Relation. Whereas parent-child relationships are used to specify interdependencies between model elements, the dependee-dependent relation is used for specifying unidirectional dependencies. An example of a dependee-

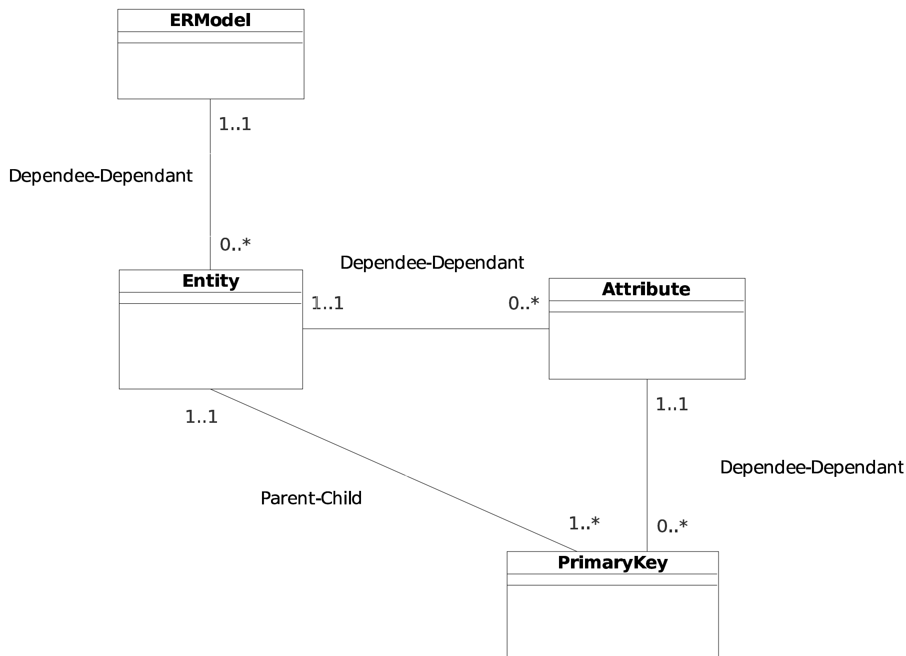

Figure 7: Example of Parent-Child and Dependee-Dependant relations

dependant relation exists between **Attribute** and **Key**. Keys cannot exist without containing an attribute, but attributes can exist without being contained in a primary key. Also, when an attribute is removed, the key containing that attribute must be removed too. This unidirectional dependency (a key depending on an atttribute) is therefore specified using the dependee-dependant relation.

Dependency Relations Influence Eachother. Notice the dependee-dependant relation between **Entity** and **Attribute** in figure 7. At first sight, this dependency relation seems to allow the existence of an **Entity** instance *without* instances of **Attribute** being attached to it because a dependee can exist without having dependants. A scenario where an **Entity** lacks instances **Attribute**, however, is forbidden by the ER metamodel because strong entities must have a primary key that is composed of some of their attributes. This metamodel anomaly, however, can *never exist in our metamodel*, and this becomes clear if we look *further than a single dependency relation*. Indeed, there must at least be one instance of **Attribute** because otherwise there could be no instance of **PrimaryKey** (which is a dependant of **Attribute**). On the other hand, the instance of **Entity** cannot exist without instances of **PrimaryKey** being attached to it because a parent-child relation requires the parent (**Entity**) to have at least one child (**PrimaryKey**). Thus, the combination of these

requirements enforces that an **Entity** must have at least one **PrimaryKey**, which must in turn be contained in at least one **Attribute**. Summarizing, a single dependee/dependant relation between **Attribute** and **Entity** may in itself be *insufficient* to guarantee model consistency, but possible inconsistencies are avoided by closely related dependency relations working on related metamodel elements.

4.2.4 Integrating Dependency Relations with the Composite Pattern

Figure 8 shows how the general Whole-Part relation of figure 6 is split into two different dependency relations. There are two additional constraints that are not visible in this figure:

- For each instance of **IPartModelElement**, the sum of its dependees (zero or more) and parents (zero or one) must be at least one. In other words, a model element can exist without a parent only if it is the dependant of at least one dependee. Therefore, the multiplicity at the parent side should be read as "exactly one parent, unless the child is already participating in a dependee/dependant relation".
- Even though instances of **IPartModelElement** may have multiple dependees, the overall tree of the model still has at most one root element. This characteristic is inherited from our conceptual definition of the whole/part relationship, as discussed in section 4.1.

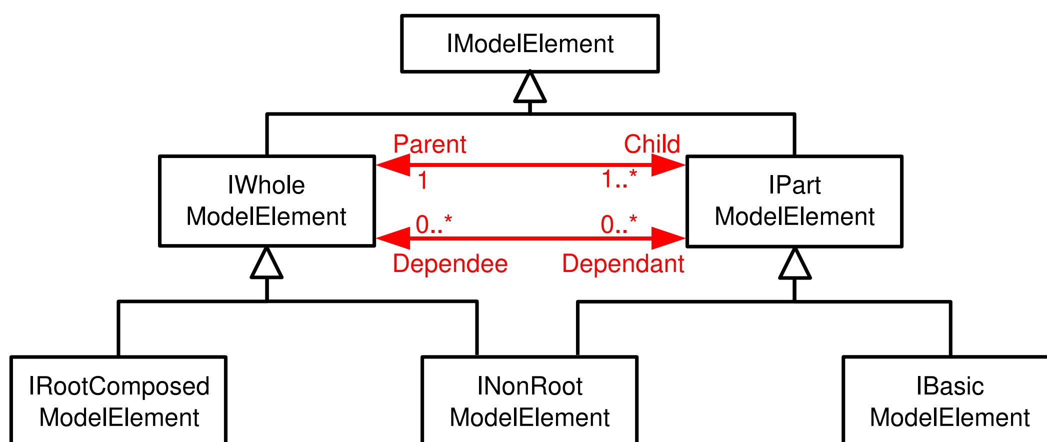

Figure 8: Integration of the Composite pattern with Parent-Child and Dependee-Dependant

4.3 Valid and Invalid States for Metamodels

Parent-Child and Dependee-Dependant relations specify structural dependencies on model elements: they ensure that the lifecycle of model elements interferes with that of closely

related model elements. The ER metamodel, for instance, uses dependency relations to ensure that no attribute or primary key can survive the deletion of the entity to which it belongs (see section 4.2). Obviously, the constraints enforced by dependency relations only work in *valid* models, i.e. in those models where all dependencies are wired correctly. This section elaborates on *model validity* and introduces the notion of *model states*.

4.3.1 Enforcing Validity Constraints

We need a way to specify whether or not a certain model is *valid*, i.e. that the model satisfies all its *invariants*. Such invariants are typically defined in the official documentation of a modeling language (e.g. the language specification). From the viewpoint of object-oriented programming, satisfying invariants means that the class representing a model element has to satisfy its *class invariants*. An example of such a (class) invariant is that a parent cannot exist without having at least one child.

Traditional solution. Following the rules of Design By Contract [14], all model elements must be valid after they are initialized. Thus, it is the responsibility of the *constructor* to ensure that a model element satisfies all of its class invariants. An advantage of this approach is that the transformation tool may assume that a model is consistent from the moment it is created. In other words, every model element is valid at all times. In the context of this project, however, this traditional appoach leads to a number of problems:

- *Circular dependencies*. Some dependencies make it impossible to ensure that the model element satisfies all of its invariants after construction. For example, creating a strong entity in the ER model requires us to create a primary key, which in turn requires at least one attribute, but this attribute is a dependant of the entity that we are creating, and must therefore contain a reference to the entity being created!
- *Incremental model instantiations*. When models are instantiated incrementally, for instance by parsing XML-like model desciptions, it is very hard to ensure that the model is consistent after each parsing step. Indeed, this approach requires descriptors to be written in such a way that each point in the file comprises a valid model, and that each parsing step extends the model in such a way that the new model is again consistent. Given the existence of circular dependencies, it is unlikely that such incrementally consistent model descriptions even exist. Even if they do, it is questionable whether such a rigid description format is desirable.

Weakening validity constraints during model instantiation. Due to the occurence of cicular dependencies complicating the stepwise instantiation of consistent models, we have decided to weaken the rules concerning model validity. In stead of requiring that each

model element is valid after construction, we require the model element to be valid *after some period of time*, as shown in figure 9. Each model element therefore has a *validity state*, which is initially set to *invalid*, meaning that the model element does not satisfy its invariants. This initial state gives us a time frame for creating other (invalid) model elements and for wiring them together until all consistency rules are satisfied. From then, the state of the objects can be changed to *valid* and this state must hold for the entire lifespan of the model element. That is, model elements are never invalidated once they become valid parts of a model.

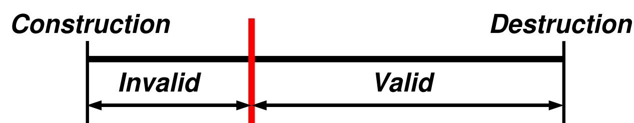

Figure 9: Model elements are granted a time frame in which they may violate invariants.

4.3.2 Example

Suppose we want to create an **Entity** with a **PrimaryKey** consisting of a list of **Attribute** instances —the dependencies to be resolved are shown in figure 7 in section 4.2. In that case, we could first create a primary key instance and this instance would reside in an *invalid* state because (1) it violates the parent/child relation with the entity for which it is the primary key and (2) it violates the dependee/dependant relation with the attributes it is composed of. By relaxing model validity requirements during model instantiation, however, we are granted a time frame for satisfying all invariants. Thus, we may create a list of attributes and instantiate an entity, add these attributes to the entity, and add a subset of these attributes to the primary key. This primary key can then be attached to the entity. Finally, we can change the state of all model elements to *valid* because all dependencies are registered correctly.

5 Pluto API Specification

This section describes the API specification of our Pluto framework for building metamodels. The API of the key interfaces for metamodeling is discussed in detail as these interfaces are extensively used in the remainder of this document. Note, however, that these interfaces constitute only a part of the Pluto framework; a second part is given in the text that discusses model transformations (see section 8).

5.1 IModelElement

The **IModelElement** interface exports operations for managing dependencies. The first part of the code constitutes operations for managing parent/child relations. It is possible to register a parent (**changeParentTo**) and check whether such a parent is a valid parent for the model element (**canHaveAsParent**). There is also an inspector for retrieving a reference to that parent (**getParent**). Notice that a parent is always an instance of **IComposedModelElement** because basic model elements (i.e. those elements implementing the **IBasicModelElement**) are not allowed to contain other model elements; they are the leaves of the model tree.

```
package aspectlab.framework.classes;

import java.util.Iterator;

/**
 * An interface of model elements.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface IModelElement{

        /**
         * Return the parent of this model element.
         *   A null reference is returned if this model element has no parent.
         * @note This model element is a dependant of its (effective) parent.
         */
        public abstract IComposedModelElement getParent();

        /**
         * Check whether this model element can have the given parent as its parent.
         * @param   parent
         *          The parent to check
         * @return | if (parent == this)
         *         |   then result == false
         */
        public abstract boolean canHaveAsParent(IComposedModelElement parent);

        /**
         * Change the parent of this model element to the given parent.
         * @param   parent
         *          The new parent for this model element.
         * @post    The new parent of this model element is the given parent.
         *          | new.getParent() == parent.
         * @post    | (parent == null) || (new parent).hasAsElement(this)
    *           | (getParent() == null) || !(new getParent()).hasAsElement(this)
         * @throws IllegalArgumentException [must]
         *          | !canHaveAsParent(parent)
         * @note    In view of later applications (e.g. refactoring) it is better
         *          to allow changes to the parent, meaning that the first clause
         *          in the exception condition is better removed. This also means
         *          changes to postconditions (removing from old parent) //todo remove note
         */
    public abstract void changeParentTo(IComposedModelElement parent);

        /**
         * Check whether this model element has the given composed model element as its
         * direct or indirect parent.
         * @param   element
         *          The element to check
         * @return | result == (getParent() == element) ||
         *         |   ( (getParent() != null) &&
         *         |      getParent().hasAsDirectOrIndirectParent(element))
         */
        public abstract boolean hasAsDirectOrIndirectParent(IComposedModelElement element);
```

```
/**
 * Return the direct dependant of this model element at the given index.
 * @param   index
 *          The index of the direct dependant to return.
 * @throws IndexOutOfBoundsException [must]
 *          | (index < 1) || (index > getNbDirectDependants())
 */
public abstract IModelElement getDirectDependantAt(int index)
                throws IndexOutOfBoundsException;

/**
 * Return the number of direct dependants of this model element.
 */
public abstract int getNbDirectDependants();

/**
 * Check whether this model element can have the given element as one of
 * its dependants.
 * @param   element
 *          The element to check
 * @return  | if ( (element == null) ||
 *          |      (!element.canHaveAsDirectDependee(this)) )
 *          | then result == false
 */
public abstract boolean canHaveAsDirectDependant(IModelElement element);

/**
 * Check whether this model element has the given model element as one of
 * its direct dependants.
 * @param   element
 *          The element to check
 * @return  | result == (element != null) && element.hasAsDirectDependee(this)
 */
public abstract boolean hasAsDirectDependant(IModelElement element);

/**
 * Check whether this model element has the given model element as one of
 * its direct parts.
 * @param   element
 *          The element to check
 * @return  | (element != null) && element.hasAsDirectWhole(this)
 */
public abstract boolean hasAsDirectPart(IModelElement element);

/**
 * Check whether this model element has the given model element as one of
 * its direct or indirect parts.
 * @param   element
 *          The element to check
 * @return  | (element != null) && element.hasAsDirectOrIndirectWhole(this)
 */
public abstract boolean hasAsDirectOrIndirectPart(IModelElement element);

/**
 * Check whether this model element has the given model element as one of its
 * direct or indirect dependants.
 * @param   element
 *          The element to check
 * @return  | result == (element != null) && element.hasAsDirectOrIndirectDependee(this)
 */
public abstract boolean hasAsDirectOrIndirectDependant(IModelElement element);

/**
 * Return an iterator over the direct parts of this model element.
 * @return  | result != null
 * @return  | if (getNbDirectDependants() > 0)
 *          |   then result.hasNext()
 * @return The iterator will return all model elements that are a direct
 *          part of this model element (i.e. all model elements E for
 *          which hasAsDirectPart(E) is true).
 */
public abstract Iterator<IModelElement> getDirectParts();

/**
 * Return an iterator over the direct dependants of this model element.
 * @return  | result != null
```

```
 * @return | if (getNbDirectDependants() > 0)
 *         |   then result.hasNext()
 * @return The iterator will return all model elements that are a direct
 *         part of this model element (i.e. all model elements E for
 *         which hasAsDirectDependant(E) is true).
 */
public abstract Iterator<IModelElement> getDirectDependants();

/**
 * Add the given dependant as another direct dependant for this model element
 * @param  dependant
 *         The dependant to add.
 * @pre    | this.canHaveAsDirectDependant(dependant)
 * @post   | new.getDirectDependantAt(getNbDirectDependants()+1) ==
 *         |           dependant
 * @post   | new .getNbDirectDependants() == getNbDirectDependants()+1
 * @note   This method is unsafe in the sense that it only registers the
 *         dependency association in one direction.
 */
public abstract void addDirectDependant(IModelElement dependant);

/**
 * Remove the given dependant as a direct dependant for this model element.
 * @param  dependant
 *         The dependant to remove.
 * @pre    | this.hasAsDirectDependant(dependant)
 * @post   | ! new.hasAsDirectDependant(dependant)
 * @post   | new.getNbDirectDependants() == getNbDirectDependants() - 1
 * @note   This method is unsafe in the sense that it only registers the
 *         dependency association in one direction.
 */
public abstract void removeDirectDependant(IModelElement dependant);

/**
 * Return the direct dependee of this model element at the given index.
 * @param  index
 *         The index of the direct dependee to return.
 * @throws IndexOutOfBoundsException
 *         | (index < 1) || (index > getNbDirectDependees())
 */
public abstract IModelElement getDirectDependeeAt(int index)
                throws IndexOutOfBoundsException;

/**
 * Return the number of direct dependees of this model element.
 */
public abstract int getNbDirectDependees();

/**
 * Check whether this model element can have the given element as one of
 * its dependees.
 * @param  element
 *         The element to check
 * @return | if (element == null)
 *         |   then result == false
 * @return | if (element == this)
 *         |   then result == false
 * @return | if ( (element != null) &&
 *         |      element.hasAsDirectOrIndirectDependee(this) )
 *         |   then result == false
 */
public abstract boolean canHaveAsDirectDependee(IModelElement element);

/**
 * Check whether this model element has the given model element as one of
 * its direct dependees.
 * @param  element
 *         The element to check
 * @return | result == (element != null && getParent() == element) ||
 *         |           ( for some I in 1..getNbDirectDependees():
 *         |                 (getDirectDependeeAt(I) == element) )
 */
public abstract boolean hasAsDirectDependee(IModelElement element);

/**
 * Check whether this model element has the given model element as one of
```

```
 * its direct wholes.
 * @param   element
 *          The element to check
 * @return  | result == hasAsDirectDependee(element) ||
 *          |            ((element != null) && (getParent()==element))
 */
public abstract boolean hasAsDirectWhole(IModelElement element);

/**
 * Check whether this model element has the given model element as one of
 * its direct or indirect wholes.
 * @param   element
 *          The element to check
 * @return  | result == hasAsDirectWhole(element) ||
 *          |            (for some modelElement in IModelElement :
 *          |               hasAsDirectWhole(modelElement) &&
 *          |               modelElement.hasAsDirectOrIndirectWhole(element))
 */
public abstract boolean hasAsDirectOrIndirectWhole(IModelElement element);

/**
 * Check whether this model element has the given model element as one of its
 * direct or indirect dependees.
 * @param   element
 *          The element to check
 * @return  | result == this.hasAsDirectDependee(element) ||
 *          |            ( for some I in 1..getNbDirectDependees():
 *          |                getDirectDependeeAt(I).
 *          |                  hasAsDirectOrIndirectDependee(element) )
 */
public abstract boolean hasAsDirectOrIndirectDependee(IModelElement element);

/**
 * Return an iterator over the direct wholes of this model element.
 * @return  | result != null
 * @return  | if (getNbDirectDependees() > 0)
 *          |   then result.hasNext()
 * @return  | if (getParent() != null)
 *          | then result.hasNext()
 * @return The iterator will return all model elements that are a direct
 *          whole of this model element (i.e. all model elements E for
 *          which hasAsDirectWhole(E) is true).
 */
public abstract Iterator<IModelElement> getDirectWholes();

/**
 * Return an iterator over the direct dependees of this model element.
 * @return  | result != null
 * @return  | if (getNbDirectDependees() > 0)
 *          |   then result.hasNext()
 * @return The iterator will return all model elements that are a direct
 *          whole of this model element (i.e. all model elements E for
 *          which hasAsDirectDependee(E) is true).
 */
public abstract Iterator<IModelElement> getDirectDependees();

/**
 * Add the given dependee as another direct dependee for this model element
 * @param   dependee
 *          The dependee to add.
 * @pre     | this.canHaveAsDirectDependee(dependee)
 * @post    | new.getDirectDependeeAt(getNbDirectDependees()+1)) == dependee
 * @post    | new.getNbDirectDependees() == getNbDirectDependees() + 1
 * @post    | (new dependee).
 *          |   getDirectDependantAt(dependee.getNbDirectDependants()+1))
 *          |           == this
 * @post    | (new dependee).getNbDirectDependants() ==
 *          |         dependee.getNbDirectDependants() + 1
 */
public abstract void addDirectDependee(IModelElement dependee);

/**
 * Remove the given dependee as a direct dependee for this model element.
 * @param   dependee
 *          The dependee to remove.
 * @pre     | this.hasAsDirectDependee(dependee)
```

```
 * @post    | ! new.hasAsDirectDependee(dependee)
 * @post    | new.getNbDirectDependees() == getNbDirectDependees() - 1
 * @post    | ! (new dependee).hasAsDirectDependant(this)
 * @post    | (new dependee).getNbDirectDependants() ==
 *          |                   dependee.getNbDirectDependants() -1
 */
public abstract void removeDirectDependee(IModelElement dependee);

/**
 * Print this transformable model element to the standard output stream.
 * @param   indent
 *          The indent in front of the printed information.
 */
public abstract void print(String indent);
}
```

5.2 IComposedModelElement

The **IModelElement** superinterface is specialized by two interfaces. The first one, **IComposedModelElement** is discussed in this section. This interface is implemented by classes that act as *containers* for model elements. Consequently, instances of **IComposedModelElement** are found at the root node and at internal nodes of a model tree. The **IComposedModelElement** interface inherits dependency management for parent/child and dependee/dependant relations from **IModelElement** and introduces a number of operations for managing the model elements it contains (among others: **addElement**, **removeElement**, and **hasAsElement**). These contained model elements are either internal nodes (implementing **IComposedModelElement**) or leaf nodes (implementing **IBasicModelElement**), and therefore, we accept all elements implementing the common ancestor, **IModelElement**.

```
package aspectlab.framework.classes;

import java.util.Iterator;

/**
 * An interface of composed model elements.
 * - A composed model element can be composed of other model elements.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface IComposedModelElement extends IModelElement, Iterable<IModelElement> {

    /**
     * Check whether this composed model element is the direct or indirect parent
     * of the given model element.
     * @param   element
     *          The element to check
     * @return  | result == (element != null) && (element.hasAsDirectOrIndirectParent(this))
     */
    public abstract boolean isDirectOrIndirectParentOf(IModelElement element);

    /**
     * Check whether this model element can have the given parent as its parent.
     * @return  | if ((parent != null) && parent.hasAsDirectOrIndirectParent(this))
     *          |    then result == false
     * @see IModelElement
     */
```

```
    public abstract boolean canHaveAsParent(IComposedModelElement parent);

    /**
     * Check whether this composed model element has the given element as one
     * of its elements.
     * @param  element
     *         The element to check.
     * @return | result == (element != null) && (element.getParent() == this)
     */
    public abstract boolean hasAsElement(IModelElement element);

    /**
     * Check whether this composed model element can have the given element as one
     * of its elements.
     * @param  element
     *         The element to check.
     * @return | if ( (element == null) ||
     *         |      (! element.canHaveAsParent(this)) )
     *         | then result == false
     */
    public abstract boolean canHaveAsElement(IModelElement element);

    /**
     * Return the number of elements of this composed model element.
     * @return | card( {element:IModelElement| this.hasAsElement(element)} )
     */
    public abstract int getNbElements();

    /**
     * Return an iterator over the elements of this composed model element.
     * @return | result != null
     * @return | if (getNbElements() > 0)
     *         |   then result.hasNext()
     * @return The iterator will return all model elements that are a direct
     *         element of this model element (i.e. all model elements E for
     *         which hasAsElement(E) is true).
     */
    public abstract Iterator<IModelElement> iterator();

    /**
     * Add the given element to this composed model element.
     * @param  element
     *         The element to add.
     * @pre    | this.canHaveAsElement(element)
     * @post   | new.hasAsElement(element)
     * @note   This method is unsafe in the sense that it only registers the
     *         composition association in one direction.
     */
    public abstract void addElement(IModelElement element);

    /**
     * Remove the given element from this composed model element.
     * @param  element
     *         The element to remove.
     * @pre    | this.hasAsElement(element)
     * @post   | ! new.hasAsElement(element)
     * @note   This method is unsafe in the sense that it only registers the
     *         composition association in one direction.
     */
    public abstract void removeElement(IModelElement element);
}
```

5.3 IBasicModelElement

Next to **IComposedModelElement**, the **IBasicModelElement** interface is a second specialization of the general **IModelElement** interface. This second interface is implemented by those classes that are found at the *leaf nodes* of the model tree. Together with **ICom-**

posedModelElement, **IBasicModelElement** is used to implement the Composite pattern, as discussed in section 4.1. Indeed, an **IComposedModelElement** may recursively contain other composed model elements, and this recursion is stopped by those composed model elements that only contain instances of **IBasicModelElement** as their children.

```
package aspectlab.framework.classes;

/**
 * An interface of basic model elements.
 *  - A basic model element has no other model elements as its components.
 *
 *  @invar This model element has at least one dependee or a parent.
 *         | getParent() != null || getNbDirectDependees() > 0
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface IBasicModelElement extends IModelElement {

}
```

5.4 IRootModelElement

The Composite pattern of section 4.1 introduced additional interfaces for fine-grained dependency management in the Pluto framework. These interfaces are not yet implemented in the current version of the tool. This is a simple yet time-consuming refactoring, so it will be completed in a newer version. In this API overview, we stick with the basic Composite pattern and specialize the **IComposedModelElement** in two subinterfaces: **IRootModelElement** and **INonRootModelElement**. The API of the former is shown here. It contains only a single operation, **canHaveAsParent**. The specification of this method requires the parent of a root node to be **null**. Indeed, root nodes cannot be contained in other model elements, so they may not have a parent.

```
package aspectlab.framework.classes;

/**
 * An interface of root model elements.
 *  - A root model element has no parent.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface IRootModelElement extends IComposedModelElement {

        /**
         * Check whether this root model element can have the given parent as its parent.
         * @return | result == (parent == null)
         * @see     IComposedModelElement
         */
        public abstract boolean canHaveAsParent(IComposedModelElement parent);

        /**
         * Check whether this root model element can have the given element
         * as one of its dependees.
         * @param  element
```

```
	 *            The element to check
	 * @return | result == false
	 * @see IModelElement
	 */
	public abstract boolean canHaveAsDirectDependee(IModelElement element);
}
```

5.5 INonRootModelElement

The **INonRootModelElement** strongly resembles the **IRootModelElement** in that they both export the same operation, **canHaveAsParent**. Their specification, however, strongly differs. Indeed, whereas an **IRootModelElement** *must not* have an effective parent (i.e. the parent must be null), an **INonRootModelElement** must have another model element as a parent in order to be valid.

```
package aspectlab.framework.classes;

/**
 * An interface of non root model elements.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 *
 *  @invar This model element has at least one dependee or a parent.
 *         | getParent() != null || getNbDirectDependees() > 0
 */
public interface INonRootModelElement extends IComposedModelElement {

}
```

6 Concrete Metamodels based on Pluto

Section 4 provided a conceptual view on the reusable features of the Pluto framework. These concepts were explained in section 5 by means of both the informal and the formal API specification of the Pluto interfaces for constructing concrete metamodels. In this section, we show how such concrete metamodels are defined starting from the general functionality that is provided by the Pluto framework. In later sections, we reuse these concrete metamodels to implement *model transformations*. We describe the Entity-Relation Metamodel in section 6.1 and the Relational Database Metamodel in section 6.2.

6.1 The Entity-Relation Metamodel

We have implemented the major part of the ER metamodel by extending the reusable concepts of the Pluto framework for metamodeling. This concrete metamodel is shown in figure 10. Entity-Relation models are represented by the **IERModel** interface. This is

the root node of an ER model tree and therefore, this interface extends the **IRootModelElement** interface. An ER model can exist without containing entities, so it maintains a dependee/dependant relations (in stead of a parent/child relation) with an **IType**. Concrete implementations of these interfaces inherit dependency management from default implementations of the **IComposedModelElement** interface, so the developer of this metamodel is no longer responsible for guaranteeing the correctness of these dependencies.

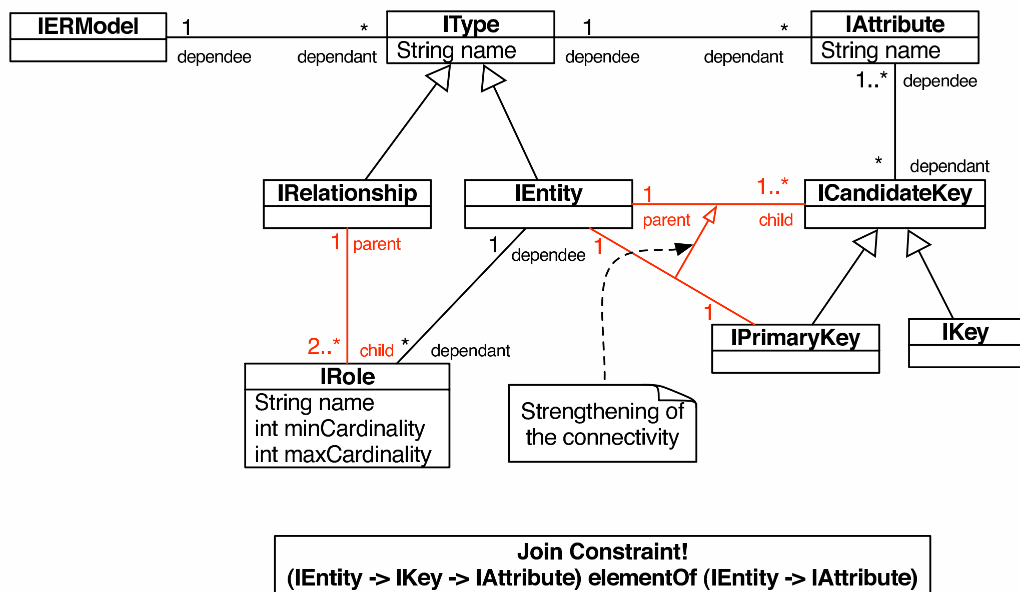

Figure 10: Concrete ER metamodel

IType extends the **INonRootModelElement** interface because its concrete realizations represent *internal nodes* of an ER model tree. On one hand, they are contained in an **IERModel**; on the other hand, they contain instances of **IAttribute**. The relation with **IAttribute** is again a dependee/dependant relation because concrete instances of **IType** (either entities or relationships) can continue to exist without containing attributes.

An example of a parent/child relation is found at **IRole**. This is a subtype of **IBasicModelElement** because roles do not contain other model elements. On the other hand, they *are* contained in an **IRelationship** and in an **IEntity** because they represent the role that an entity plays in a certain relationship.

Join constraint. Next to depicting dependency relations, figure 10 also shows how a *join constraint* is introduced to increase the semantic information of the metamodel. This con-

straint requires that the attributes belonging to an entity's key must be attributes of that entity. Such constraints are specific to the ER metamodel, so they cannot be enforced at the level of the Pluto framework. Also, they cannot be shown in the model by simply drawing associations between the interfaces of the metamodel. Therefore, we have added them as an OCL expression.

Constraining dependencies. Figure 10 also shows how parent-child relations can be strengthened by concrete metamodels. The default semantics of this dependency relation state that a parent can have multiple children, but these semantics are refined at the level of the **IEntity** interface. Indeed, the concrete metamodel constrains the number of **IPrimaryKey** instances to be exactly one. Thus, although it is possible to have multiple candidate keys, it is not allowed to have more than one primary key at the same time. Strengthening such dependency relations requires intervention of the metamodel developer because such rigorous semantics are metamodel-specific.

Evaluation – Pluto and the ER metamodel.

There are a number of ways in which Pluto eases the introduction of the ER metamodel. First, the general dependency relations of Pluto allow for easy wiring between different metamodel constructs. Developers only have to strengthen some dependencies when the metamodel requires so (e.g. the parent/child dependency between **IEntity** and its **IPrimaryKey**). Another advantage is that model validity and model composition are provided at a reusable level. By defining **IRole** as an extension of **IBasicModelElement**, for instance, developers ensure that no **IRole** can ever contain other model elements. Also, by defining **IERModel** as a subtype of **IRootModelElement**, developers enforce that no model element can ever contain an **IERModel** element as one of its children or dependants.

6.2 The Relational Database Metamodel

It is important that the reader is introduced to two different metamodels before starting the part of this text that is concerned with *cross-model transformations*. In the remainder of this text, concrete instances of the ER metamodel will be transformed into concrete instances of the RDB metamodel. Therefore, we also introduce the RDB metamodel, which is shown in figure 11. The **IDatabase** interface is comparable to the **IERModel** interface in the ER metamodel because they both extend the **IRootModelElement** interface. In fact, further in this text, we will show how instances of **IERModel** can be transformed to instances of **IDatabase**.

IDatabase maintains a dependee/dependant relation with **IRelation**. This concept com-

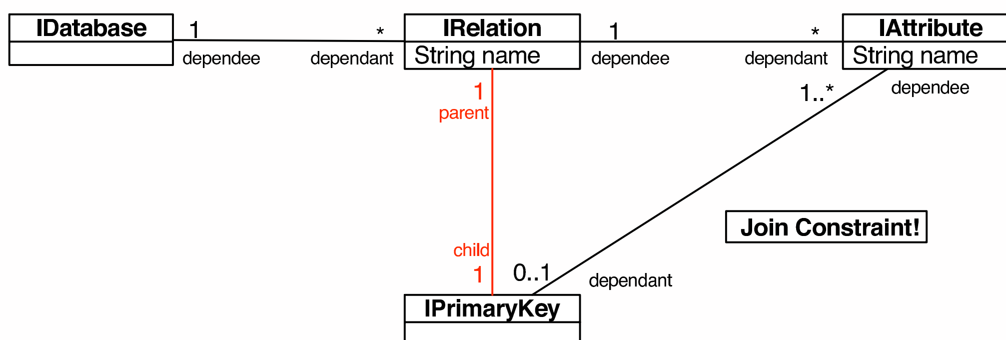

Figure 11: Concrete RDB metamodel

pares to an **IEntity** in ER and represents a database table. Notice how the parent/child relation is again strengthened between **IRelation** and **IPrimaryKey**. Augmenting these semantics is the task of the developers because such strengthenings are model-specific; they cannot be abstracted to the level of the Pluto framework. Also, note how a similar join constraint is defined for the metamodel shown in figure 11: the attributes of an entity's primary key must belong to that entity.

From Metamodeling to Model Transformations

At this point, we have explained how Pluto provides reusable concepts for implementing concrete metamodels. These concepts were discussed in section 4, the Pluto API specification for metamodeling was given in section 5 and concrete realizations of the ER and RDB metamodels were shown in the current section. The remainder of this text shows how models, designed as instances of these metamodels, can be transformed between different modeling languages. Again, Pluto will play a key role in these model transformations by providing reusable model transformation concepts.

7 Pluto Support for Model Transformations

Similar to the first part of the text, we explain a number of reusable constructs offered by the framework in order to speed up the implementation of model transformations. The concepts introduced in this part of the text heavily rely on support offered by Pluto for building metamodels, including composition, dependency relations, and valid/invalid states. In this

section, we show how the interfaces introduced in section 5 are extended so as to make models transformable. The API of these specific interfaces is presented in section 8, whereas section 9 shows how transformations are realized in the concrete metamodels of section 6.

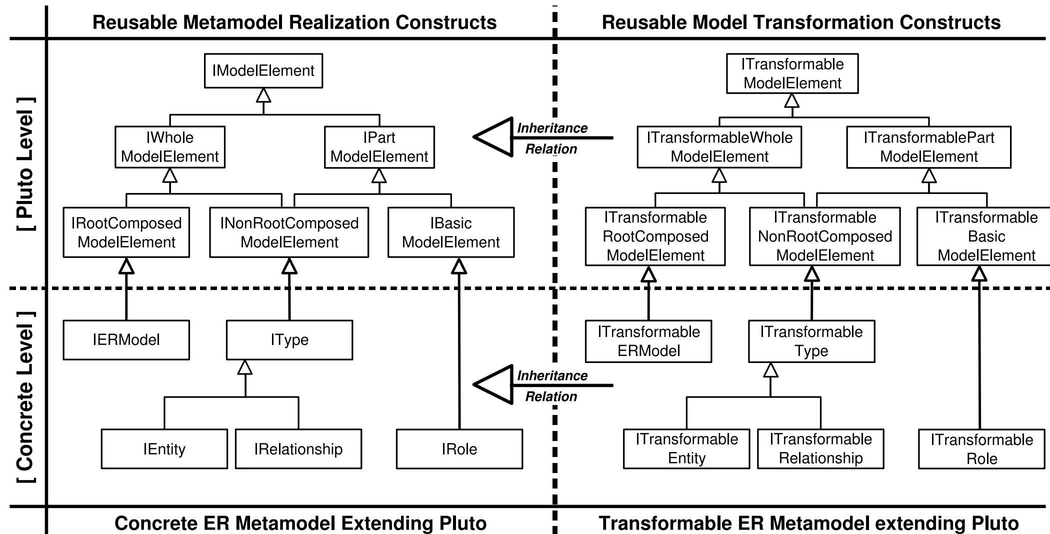

Figure 12: Overview of the Pluto framework for metamodeling

The left part of figure 12 shows the interfaces offered by Pluto for building concrete metamodels. These interfaces are extended by special *transformation interfaces*, as shown in the right part of the figure. There is a one-to-one mapping between general modeling interfaces and specialized transformation interfaces. **IModelElement**, for instance, is a supertype of **ITransformableModelElement**, and **INonRootComposedModelElement** is a supertype of **ITransformableNonRootComposedModelElement**.

Making Metamodels Transformable. Plain metamodels without transformable behaviour are created by introducing classes that implement the interfaces shown in the left part of figure 12. These classes represent the concrete modeling constructs of a language (e.g. the entities and attributes of the ER modeling language described in section 6). In order to attach transformable behaviour, these classes can be *subclassed* where each subclass implements the corresponding *transformation interface*. In figure 12, for instance, an **Entity** class can be introduced as an implementation of the **IEntity** interface. Transformable behaviour can then be attached to this **Entity** by subclassing it and by implementing the **ITransformableEntity** interface.

8 Pluto Model Transformation API

Given the basic idea that transformable behaviour is added by implementing special *transformation interfaces*, we now show the complete API reference of the Pluto interfaces that offer transformable functionality. These interfaces inherit common functionality such as dependency management and model element composition from their superinterfaces (their API is given in section 5) and introduce operations that support the generic transformation algorithm discussed in section 10.

8.1 ITransformableElement

As shown in figure 12, there is a one-to-one mapping between *general interfaces* for model elements, and *specific interfaces* that introduce transformable behaviour for model elements. **ITransformableModelElement**, for instance, is the transformable complement of **IModelElement**. Being a subtype of **IModelElement**, **ITransformableModelElement** inherits dependency and composition management for model elements. At the same time, this subinterface introduces operations for *transforming* the model element (**makeTransformation**, **setTransformation**, and **executeTransformation**).

```
package aspectlab.framework.transformables;

import aspectlab.framework.classes.*;
import aspectlab.framework.strategies.IStrategy;

/**
 * An interface of transformable elements.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface ITransformableElement
                extends IModelElement {

        /**
         * Return the strategy for this transformable element.
         */
        public abstract IStrategy getStrategy();

        /**
         * Return the target element that is generated by the strategy
         * of this transformable element.
         * @pre | isTransformed()
         * @return | result == getStrategy().getTargetElement()
         */
        public abstract IModelElement getTargetElement();

        /**
         * Check whether this transformable element can have the given
         * strategy as its strategy.
         * @return | if (strategy == null)
         *         | then result == false
         */
        public abstract boolean canHaveAsStrategy(IStrategy strategy);

        /**
         * Set the given strategy as the strategy for this
         * transformable element.
         * @param strategy
```

```
 *             The strategy to be registered.
 * @pre        | canHaveAsStrategy(strategy)
 * @pre        | (strategy.getSourceElement() == this)
 * @post       | new.getStrategy() == strategy
 */
public abstract void setStrategy(IStrategy strategy);

/**
 * Make a strategy for this transformable element.
 * @post    | new.getStrategy() != null
 * @post    | new.getStrategy().getSourceElement() == this
 */
public abstract void makeStrategy();

/**
 * Execute the strategy for this transformable element.
 * @pre     | canBeTransformed()
 * @post    | new.isTransformed()
 * @effect  | getStrategy().execute()
 */
public abstract void executeStrategy();

/**
 * Undo the transformation for this transformable element.
 * @pre   | canBeUndoTransformed()
 * @post  | !isTransformed()
 * @effect  | getStrategy().undo()
 */
public abstract void undoStrategyExecution();

/**
 * Check whether this transformable element can be transformed.
 * @return  | if (getStrategy() == null)
 *          | then result == false
 * @return  | if (isTransformed())
 *          | then result == false
 * @return This transformable element can be transformed if all
 *          transformable elements it is existential dependent of
 *          are transformed.
 *          | if (for some element in ITransformableElement:
 *          |         hasAsDirectWhole(element) &&
 *          |         ! element.isTransformed())
 *          | then result == false
 */
public abstract boolean canBeTransformed();

/**
 * Check whether the transformation of this transformable element
 * can be undone.
 * @return  | if (getStrategy() == null)
 *          | then result == false
 * @return  | if (! isTransformed())
 *          | then result == false
 * @return The transformation of this transformable element can be
 *          undone if no transformable element that existential
 *          depends on this transformable element is transformed.
 *          | if (for some element in ITransformableElement:
 *          |         hasAsDirectPart(element) &&
 *          |         element.isTransformed())
 *          | then result == false
 */
public abstract boolean canBeUndoTransformed();

/**
 * Transform this transformable element.
 * @pre TODO canBe...
 * @post | isTransformed()
 * @effect | for some element in ITransformableElement:
 *           | element.makeStrategy();
 *           | element.executeStrategy()
 * @note The above effect-clause uses the ";"-operator : the
 *        expression before the operator must be realised before the
 *        expression after the operator
 * @post | for all element in ITransformableElement:
 *        |    if (hasAsDirectOrIndirectWhole(element))
 *        |    then element.isTransformed()
```

```
     * @post | for all element in ITransformableElement:
     *       |    if (hasAsDirectOrIndirectPart(element))
     *       |    then element.isTransformed()
     * @note This method is the framework method that is responsible for
     *       the transformation order of all transformable model elements.
     *       The effective transformation action is done by executeStrategy().
     * TODO Is deze specificatie niet te niet-deterministisch?
     */
    public abstract void transform();

    /**
     * Undo the transformation of this transformable element.
     * @pre TODO canBe...
     * @post | ! isTransformed()
     * @post | for all element in ITransformableElement:
     *       |    if (hasAsDirectOrIndirectPart(element))
     *       |    then ! element.isTransformed()
     * @note This method is the framework method that is responsible for
     *       the undo transformation order of all transformable model elements.
     *       The effective transformation action is done by undoTransform().
     */
    public abstract void undoTransformation();

    /**
     * Check whether this transformable element is transformed.
     */
    public abstract boolean isTransformed();

    /**
     * Change the transformation state to the given state.
     * @post | new.isTransformed() == newTransformedState
     *
     */
    public abstract void setTransformed(boolean newTransformedState);

  /**
   * Return the parent of this transformable model element.
   * @see IModelElement
   */
  public IComposedTransformableElement getParent();

  /**
   * Return the direct dependant of this model element at the given index.
   * @see IModelElement
   */
  public ITransformableElement getDirectDependantAt(int index) throws IndexOutOfBoundsException;

  /**
     * Check whether this transformable element can have the given parent as its parent.
     * @return | if ( (parent != null) &&
     *         |        (! (parent instanceof IComposedTransformableElement)) )
     *         |    then result == false
     * @see      IModelElement
     */
    public abstract boolean canHaveAsParent(IComposedModelElement parent);

  /**
     * Check whether this transformable element can have the given element as one of
     * its dependants.
     * @return | if (! (element instanceof ITransformableElement) )
     *         |    then result == false
     * @see      IModelElement
     */
    public abstract boolean canHaveAsDirectDependant(IModelElement element);

  /**
     * Check whether this transformable element can have the given element as one of
     * its dependees.
     * @return | if (! (element instanceof ITransformableElement) )
     *         |    then result == false
     * @see      IModelElement
     */
    public abstract boolean canHaveAsDirectDependee(IModelElement element);

}
```

8.2 IBasicTransformableElement

IBasicTransformableElement complements **IBasicModelElement** with transformable behaviour. The transformation of basic model elements requires no additional operations, so it is left empty.

```
package aspectlab.framework.transformables;

import aspectlab.framework.classes.IBasicModelElement;

/**
 * An interface of basic transformable elements.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface IBasicTransformableElement extends ITransformableElement,
                IBasicModelElement {

}
```

8.3 IComposedTransformableElement

Next to **IBasicModelElement**, the Pluto framework for metamodeling also introduced **IComposedModelElement**. The complement of this second interface is **IComposedTransformableElement**. This interface introduces no additional operations with respect to **ITransformableElement**, but it tailors the specification of some operations to suit the needs of composed model element transformations.

```
package aspectlab.framework.transformables;

import aspectlab.framework.classes.IComposedModelElement;
import aspectlab.framework.classes.IModelElement;

/**
 * An interface of composed transformable model elements.
 *  - This is an interface at level M2.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface IComposedTransformableElement extends ITransformableElement,
                IComposedModelElement {

        /**
         * Check whether this composed transformable element can have the given element
         * as one of its elements.
         * @return | if (! (element instanceof ITransformableElement))
         *         |   then result == false
         * @return | if (! getSourceElement().canHaveAsElement(element))
         *         | then result == false
         */
        public abstract boolean canHaveAsElement(IModelElement element);

}
```

8.4 INonRootTransformableElement

The **INonRootTransformableElement** interface extends the **INonRootModelElement** interface of the Pluto framework for metamodeling, and extends the **IComposedTransformableElement** interface from the Pluto transformation framework. It introduces no additional operations.

```
package aspectlab.framework.transformables;

import aspectlab.framework.classes.INonRootModelElement;

/**
 * An interface of non-root transformable elements.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface INonRootTransformableElement extends INonRootModelElement,
                IComposedTransformableElement {

}
```

8.5 IRootTransformableElement

The **IRootTransformableElement** interface extends the **IRootModelElement** interface of the Pluto framework for metamodeling, and extends the **IComposedTransformableElement** interface from the Pluto transformation framework. It introduces no additional operations.

```
package aspectlab.framework.transformables;

import aspectlab.framework.classes.IRootModelElement;

/**
 * An interface of root transformable elements.
 *  - This is an interface at level M2.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public interface IRootTransformableElement extends
                IComposedTransformableElement, IRootModelElement {

}
```

8.6 TransformableElementImpl

The **ITransformableElement** interface, which is found at the top of the hierarchy of the Pluto framework for model transformations, has a default implementation called **IRoot-**

TransformableElement. The specification and the implementation of this class are shown below.

```
package aspectlab.framework.transformables;

import java.util.Iterator;

import aspectlab.framework.classes.*;
import aspectlab.framework.strategies.IStrategy;

/**
 * A class implementing the interface of transformable elements.
 *
 * @author Eric Steegmans
 * @author Geert Delanote
 */
public abstract class TransformableElementImpl implements ITransformableElement {

	/**
	 * Initialize this new transformable model element with given underlying model element,
	 * and with no generated transformation yet.
	 * @param   modelElement
	 *          The model element underlying this new transformable model element.
	 * @post    | new.getModelElement() == element.
	 * @post    | if (modelElement.getParent() != null)
	 *          |   then (new modelElement.getParent()).hasAsElement(this)
	 * @post    | if (modelElement.getParent() != null)
	 *          |   then (! (new modelElement.getParent()).hasAsElement(modelElement))
	 * @post    | for each dependee in modelElement.getDirectDependees():
	 *          |   (new dependee.getModelElement()).hasAsDirectDependant(this)
	 * @post    | for each dependee in modelElement.getDirectDependees():
	 *          |   (! (new dependee.getModelElement()).hasAsDirectDependant(modelElement))
	 * @post    | new.getStrategy() == null
	 * @throws  IllegalArgumentException [must]
	 *          | ! canHaveAsModelElement(element)
	 * @note    It is important to have the parent of the given model element
	 *          referencing the newly created transformable element, and no
	 *          longer the given model element. Recall that the given model
	 *          element must already reference a transformable element as
	 *          its parent.
	 */
	protected TransformableElementImpl(IModelElement modelElement)
			throws IllegalArgumentException {
		if (!canHaveAsModelElement(modelElement))
			throw new IllegalArgumentException("Illegal model element!");
		this.modelElement = modelElement;
		if (modelElement.getParent() != null) {
			modelElement.getParent().removeElement(modelElement);
			modelElement.getParent().addElement(this);
		}
		Iterator<IModelElement> dependeesIterator = modelElement.getDirectDependees();
		while (dependeesIterator.hasNext()) {
			IModelElement nextDependee = dependeesIterator.next();
			nextDependee.removeDirectDependant(modelElement);
			nextDependee.addDirectDependant(this);
		}
	}

	/**
	 * Return the parent of this transformable model element.
   * @return | result == (IComposedTransformableElement) getModelElement().getParent()
	 * @see IModelElement
	 */
	public IComposedTransformableElement getParent() {
		return (IComposedTransformableElement) getModelElement().getParent();
	}

	/**
	 * Check whether this transformable element can have the given parent as its parent.
   * @return | if (! getModelElement().canHaveAsParent(parent))
	 *          | then result == false
	 * @see     ITransformableElement
	 */
	public boolean canHaveAsParent(IComposedModelElement parent) {
```

```
		return ((parent == null) || (parent instanceof IComposedTransformableElement))
			&& getModelElement().canHaveAsParent(parent);
	}

	/**
	 * Check whether this transformable element has the given model element as its
	 * direct or indirect parent.
	 * @see	IModelElement
	 */
	public boolean hasAsDirectOrIndirectParent(IComposedModelElement element) {
		return getModelElement().hasAsDirectOrIndirectParent(element);
	}

	/**
	 * Change the parent of this transformable element to the given parent.
	 * @see	IModelElement
	 */
	public void changeParentTo(IComposedModelElement parent) {
		if (!canHaveAsParent(parent)) throw new IllegalArgumentException("Illegal parent!");
	IComposedModelElement oldParent = getParent();
	if (oldParent!=null) {
	    oldParent.removeElement(this);
	    oldParent.addElement(getModelElement());
	}
	getModelElement().changeParentTo(parent);
		if (parent != null) {
			parent.removeElement(getModelElement());
			parent.addElement(this);
		}
	}

/**
	 * Return the direct dependant of this transformable model element
	 * at the given index.
	 * @param	index
	 *		The index of the direct dependant to return.
	 * @return | result == (ITransformableElement) getModelElement().
	 *		|				getDirectDependantAt(index)
	 * @throws IndexOutOfBoundsException
	 *		| (index < 1) || (index > getNbDirectDependants())
	 */
public ITransformableElement getDirectDependantAt(int index)
			throws IndexOutOfBoundsException {
		return (ITransformableElement) getModelElement().getDirectDependantAt(
			index);
	}

	/**
	 * Return the number of direct dependants of this transformable element.
 * @return | result == getModelElement().getNbDirectDependants()
	 * @see	IModelElement
	 */
	public int getNbDirectDependants() {
		return getModelElement().getNbDirectDependants();
	}

	/**
	 * Check whether this transformable element can have the given element as one of
	 * its dependants.
 * @return | if (! (getModelElement().canHaveAsDirectDependant(element)))
	 *		| then result == false
	 * @see	ITransformableElement
	 */
	public boolean canHaveAsDirectDependant(IModelElement element) {
		return (element instanceof ITransformableElement)
			&& getModelElement().canHaveAsDirectDependant(element);
	}

	/**
	 * Check whether this transformable element has the given model element as one of
	 * its direct dependants.
 * @return | getModelElement().hasAsDirectDependant(element)
	 * @see	IModelElement
	 */
	public boolean hasAsDirectDependant(IModelElement element) {
		return getModelElement().hasAsDirectDependant(element);
```

```
	}

	/**
	 * Check whether this transformable element has the given model element as one of its
	 * direct or indirect dependants.
* @return | getModelElement().hasAsDirectOrIndirectDependant(element)
	 * @see     IModelElement
	 */
	public boolean hasAsDirectOrIndirectDependant(IModelElement element) {
		return getModelElement().hasAsDirectOrIndirectDependant(element);
	}

	/**
	 * Check whether this transformable element has the given model element as one of
	 * its direct or indirect parts.
* @return | getModelElement().hasAsDirectOrIndirectPart(element)
	 * @see IModelElement
	 */
	public boolean hasAsDirectOrIndirectPart(IModelElement element){
		return getModelElement().hasAsDirectOrIndirectPart(element);
	}

	/**
	 * Return an iterator over the direct dependants of this transformable element.
	 * @see     IModelElement
	 */
	public Iterator<IModelElement> getDirectDependants() {
		return getModelElement().getDirectDependants();
	}

	/**
	 * Add the given dependant as another direct dependant for this model element
	 * @see     IModelElement
	 */
	public void addDirectDependant(IModelElement dependant) {
		//assert needs to be repeated because there is no dynamic binding
		//between ModelElementImpl and TransformableElementImpl
		assert this.canHaveAsDirectDependant(dependant);
		getModelElement().addDirectDependant(dependant);
	}

	/**
	 * Remove the given dependant as a direct dependant for this model element
	 * @see     IModelElement
	 */
	public void removeDirectDependant(IModelElement dependant) {
		getModelElement().removeDirectDependant(dependant);
	}

	/**
	 * Return the direct dependee of this transformable element at the given index.
* @param   index
	 *          The index of the direct dependee to return.
	 * @return | result == (ITransformableElement) getModelElement().getDirectDependeeAt(index)
	 * @throws IndexOutOfBoundsException
	 *          | (index < 1) || (index > getNbDirectDependees())
	 * @see     IModelElement
	 */
	public ITransformableElement getDirectDependeeAt(int index)
			throws IndexOutOfBoundsException {
		return (ITransformableElement) getModelElement().getDirectDependeeAt(
				index);
	}

	/**
	 * Return the number of direct dependees of this transformable element.
* @return | result == getModelElement().getNbDirectDependees()
	 * @see     IModelElement
	 */
	public int getNbDirectDependees() {
		return getModelElement().getNbDirectDependees();
	}

	/**
	 * Check whether this transformable element can have the given element as one of
	 * its dependees.
```

```
 * @return | if (! (getModelElement().canHaveAsDirectDependee(element)))
      *          | then result == false
      * @see     ITransformableElement
      */
     public boolean canHaveAsDirectDependee(IModelElement element) {
             return (element instanceof ITransformableElement)
                     && getModelElement().canHaveAsDirectDependee(element);
     }

     /**
      * Check whether this transformable element has the given model element as one of
      * its direct dependees.
      * @see     IModelElement
      */
     public boolean hasAsDirectDependee(IModelElement element) {
             return getModelElement().hasAsDirectDependee(element);
     }

     /**
      * Check whether this transformable element has the given model element as one of its
      * direct or indirect dependees.
      * @see     IModelElement
      */
     public boolean hasAsDirectOrIndirectDependee(IModelElement element) {
             return getModelElement().hasAsDirectOrIndirectDependee(element);
     }

     /**
      * Return an iterator over the direct dependees of this transformable element.
      * @see     IModelElement
      */
     public Iterator<IModelElement> getDirectDependees() {
             return getModelElement().getDirectDependees();
     }

     /**
      * Add the given dependee as another direct dependee for this model element.
      * @see     IModelElement
      */
     public void addDirectDependee(IModelElement dependee) {
             //assert needs to be repeated because there is no dynamic binding
             //between ModelElementImpl and TransformableElementImpl
             assert this.canHaveAsDirectDependee(dependee);
             getModelElement().addDirectDependee(dependee);
     dependee.removeDirectDependant(getModelElement());
     dependee.addDirectDependant(this);
}

     /**
      * Remove the given dependee as a direct dependee for this transformable
      * model element
      * @see     IModelElement
      */
     public void removeDirectDependee(IModelElement dependee) {
             getModelElement().removeDirectDependee(dependee);
     }

     /**
      * Return the model element underlying this transformable model element.
      */
     public IModelElement getModelElement() {
             return this.modelElement;
     }

     /**
      * Check whether this transformable element can have the given element
      * as its underlying model element.
      * @return | if (element == null)
      *         |   then result == false
      * @return | if ( element.getParent() != null) &&
      *         |     (! (element.getParent() instanceof ITransformableElement))
      *         |   then result == false
      * @return | if (for some dependee in element.getDirectDependees():
      *         |         (! (dependee instanceof ITransformableElement))
      *         |   then result == false
      * @note    It is important that the modelelement has a transformable element
```

```
 *           as its parent.
 */
public boolean canHaveAsModelElement(IModelElement element) {
        if (element == null)
                return false;
        Iterator<IModelElement> dependeesIterator = element.getDirectDependees();
        while (dependeesIterator.hasNext())
                if (! (dependeesIterator.next() instanceof ITransformableElement))
                        return false;
        return (element.getParent() == null) ||
                    (element.getParent() instanceof ITransformableElement);
}

/**
 * Variable referencing the model element to which this transformable element
 * delegates.
 */
private final IModelElement modelElement;

/**
 * Return the strategy generated for this transformable element.
 */
public IStrategy getStrategy() {
        return this.strategy;
}

/**
 * Return the target element that is generated by the strategy
 * of this transformable element.
 * @see ITransformableElement
 */
public final IModelElement getTargetElement(){
        return getStrategy().getTargetElement();
}

/**
 * Set the given strategy as the strategy generated for this
 * transformable element.
 * @see      ITransformableElement
 */
public void setStrategy(IStrategy strategy) {
        assert canHaveAsStrategy(strategy);
        assert (strategy.getSourceElement() == this);
        this.strategy = strategy;
}

/**
 * Execute the transformation for this transformable element.
 * @see      ITransformableElement
 */
public final void transform() {
        //1. Transform all elements where this element depends on
        Iterator<IModelElement> wholes = getDirectWholes();
        while (wholes.hasNext()){
                ITransformableElement nextWhole = ((ITransformableElement) wholes.next());
                if (!nextWhole.isTransformed()){
                        nextWhole.transform();
                }
        }
        if (!isTransformed()){
                //2. Make a strategy for yourself
                this.makeStrategy();
                //3. Transform yourself
                this.executeStrategy();
                //4. Transform your parts (dependants + children)
                Iterator<IModelElement> parts = getDirectParts();
                while (parts.hasNext()){
                        ((ITransformableElement) parts.next()).transform();
                }
        }
}

/**
 * Variable referencing the strategy generated for this transformable element.
 */
private IStrategy strategy;
```

```java
/**
 * Return an iterator over the direct parts of this transformable element.
 * @see IModelElement
 */
public Iterator<IModelElement> getDirectParts() {
        return getModelElement().getDirectParts();
}

/**
 * Return an iterator over the direct wholes of this transformable element.
 * @see IModelElement
 */
public Iterator<IModelElement> getDirectWholes() {
        return getModelElement().getDirectWholes();
}

/**
 * Check whether this transformable model element has the given model
 * element as one of its direct or indirect wholes.
 * @see IModelElement
 */
public boolean hasAsDirectOrIndirectWhole(IModelElement element) {
        return getModelElement().hasAsDirectOrIndirectWhole(element);
}

/**
 * Check whether this transformabel model element has the given model
 * element as one of its direct parts.
 * @see IModelElement
 */
public boolean hasAsDirectPart(IModelElement element) {
        return getModelElement().hasAsDirectPart(element);
}

/**
 * Check whether this transformable model element has the given model
 * element as one of its direct wholes.
 * @see IModelElement
 */
public boolean hasAsDirectWhole(IModelElement element) {
        return getModelElement().hasAsDirectWhole(element);
}

/**
 * Check whether this transformable element can be transformed.
 * @see ITransformableElement
 */
public boolean canBeTransformed() {
        if ( (getStrategy() == null) ||
             isTransformed() ){
                return false;
        }
        Iterator<IModelElement> directWholes = getDirectWholes();
        while (directWholes.hasNext()){
                if (!((ITransformableElement)directWholes.next()).isTransformed()){
                        return false;
                }
        }
        return true;
}

/**
 * Check whether the transformation of this transformable element
 * can be undone.
 * @see ITransformableElement
 */
public boolean canBeUndoTransformed() {
        if ( (getStrategy() == null) ||
                  (! isTransformed()) ){
                        return false;
                }
                Iterator<IModelElement> directParts = getDirectParts();
                while (directParts.hasNext()){
                        if (((ITransformableElement)directParts.next()).isTransformed()){
                                return false;
```

```
                        }
                }
                return true;
        }

        /**
         * Check whether this transformable element can have the given
         * strategy as its strategy.
         * @see ITransformableElement
         */
        public boolean canHaveAsStrategy(IStrategy strategy) {
                return strategy != null;
        }

        /**
         * Check whether this transformable element is transformed.
         * @see ITransformableElement
         */
        public boolean isTransformed() {
                return isTransformed;
        }

        private boolean isTransformed;

        /**
         * Change the transformation state to the given state.
         *
         * @see ITransformableElement
         */
        public void setTransformed(boolean newTransformedState) {
                this.isTransformed = newTransformedState;
        }

        /**
         * Execute the strategy for this transformable element.
         * @see ITransformableElement
         */
        public final void executeStrategy() {
                assert this.canBeTransformed();
                getStrategy().execute();
                setTransformed(true);

        }

        /**
         * Undo the strategy for this transformable element.
         * @see ITransformableElement
         */
        public final void undoStrategyExecution() {
                assert canBeUndoTransformed();
                getStrategy().undo();
                setTransformed(false);
        }

        /**
         * Undo the transformation of this transformable element.
         * @post | ! isTransformed()
         * @post | for all element in ITransformableElement:
         *       |    if (hasAsDirectOrIndirectPart(element))
         *       |    then ! element.isTransformed()
         * @see ITransformableElement
         */
        public void undoTransformation() {
                if (isTransformed()){
                        //1. Undo the transformation of your parts (dependants + children)
                        Iterator<IModelElement> parts = getDirectParts();
                        while (parts.hasNext()){
                                ((ITransformableElement) parts.next()).undoTransformation();
                        }
                        //2. Undo the the transformation of this element
                        this.undoStrategyExecution();
                }
        }

        /**
         * Print this model element to the standard output stream.
```

```
     * @param   indent
     *          The indent in front of the printed information.
     */
    public void print(String indent) {
        System.out.println(indent + this);
        indent += "      ";
        if (getNbDirectDependees() > 0) {
            System.out.println(indent + getNbDirectDependees() + " Dependees");
            Iterator<IModelElement> dependeesIterator = getDirectDependees();
            while (dependeesIterator.hasNext()) {
                System.out.println(indent + "  " + dependeesIterator.next());
            }
        }
        if (getNbDirectDependants() > 0) {
            System.out.println(indent + getNbDirectDependants() + " Dependants");
            Iterator<IModelElement> dependantsIterator = getDirectDependants();
            while (dependantsIterator.hasNext()) {
                System.out.println(indent + "  " + dependantsIterator.next());
            }
        }
    }

    /**
     * Return a textual representation of this Transformable Element.
     * @return | result.contains("Transformable"+getModelElement().toString())
     */
    public String toString(){
        return "Transformable "+ this.getModelElement().toString();
    }
}
```

9 Concrete Realizations of Transformable Metamodels

In this section, we reconsider the concrete ER metamodel introduced in section 6.1 and show how this metamodel can be extended in order to execute model transformations. Keeping in mind the general objective of this project –providing support for cross-model transformations– we also show how concrete Entity-Relation models are transformed to concrete Relational Database models.

9.1 The Need for Multiple Inheritance

Making metamodels transformable requires us to integrate concepts for metamodeling (e.g. implementations of the **IModelElement** interface) with concepts for model transformations (e.g. implementations of the **ITransformableModelElement** interface). Figure 13 shows that this leads to a situation where multiple inheritance is required.

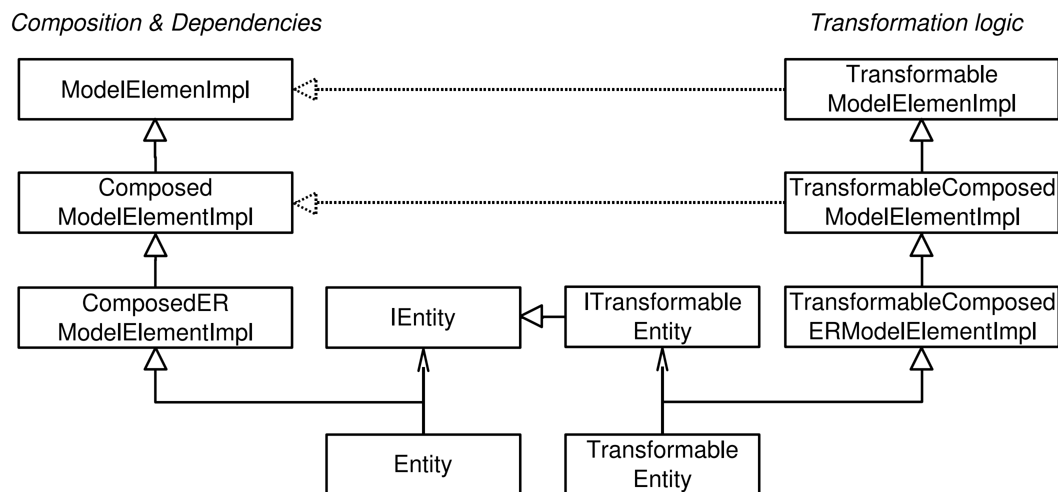

Figure 13: Classes representing transformable model elements should inherit from two superclasses (dotted arrows are not implemented)

Consider the **Entity** class, which is a concrete realization of the **IEntity** interface of the ER metamodel. This **Entity** is a subclass of **ComposedERModelElementImpl**, which introduces general functionality for composed elements in the ER model. **ComposedERModelElement** in turn inherits from **ComposedModelElementImpl**, and this class is defined at the level of the Pluto framework as an extension of **ModelElementImpl**. This first inheritance branch provides general functionality for managing dependencies and for composing model elements, as discussed in section 4.

The right part of the figure shows how an **Entity** can be made transformable by implementing the **ITransformableEntity** interface. This interface is a subtype of **IEntity**. Again, there is a default implementation available for transformable, composed ER model elements (**TransformableERModelElementImpl**). This class inherits from the Pluto framework class **TransformableComposedModelElementImpl**, which is an extension of **TransformableElementImpl**. This second inheritance branch provides reusable functionality for executing model transformations, as will be explained in section 10. **TransformableEntity** needs functionality for managing model dependencies (from the first branch) and requires support for executing transformations (from the second branch). Therefore, **TransformableEntity** should inherit from two classes at the same time: **ComposedERModelElementImpl** and **TransformableComposedModelElementImpl**.

In Java, however, multiple inheritance is not supported, so we cannot reuse *both* dependency management (defined at **IComposedModelElement** and implemented by **ComposedModelElementImpl**) and transformation logic (defined at **ITransformableComposedModelElement** and implemented by **TransformableComposedModelElementImpl**). At first sight, this problem could be solved by introducing the dotted inheritance arrows, shown in figure 13. It is indeed possible to let **TransformableModelElementImpl** extend **ModelElementImpl**. The second arrow, however, introduces similar problems because **TransformableComposedModelElementImpl** must inherit from the default implementation **TransformableModelElementImpl** *and* from **ComposedModelElementImpl**. Thus, multiple inheritance is again necessary in order to avoid massive code duplication.

The lack of multiple inheritance in Java can be circumvented in two different ways. On one hand, we could directly integrate transformation logic into the metamodeling concepts. This is not a good solution, however, as it violates some of the design goals mentioned in section 2. Indeed, the separation of concerns between constructing metamodels and transforming instances of those metamodels would be lost, as such decreasing the reusability of our metamodels in tools where model transformations are not required. On the other hand, we could circumvent the lack of multiple inheritance in Java by applying *design patterns*. In this project, we have chosen to follow the second path, and we have used the Decorator pattern [9] to make metamodels transformable without violating our design goals concerning obliviousness. The use of this Decorator pattern is described in a separate section.

9.2 The Decorator Pattern as an Extension Mechanism

The basic idea of the Decorator pattern is shown in figure 14. It shows how an **Element** (this could be an **Entity**) implements general functionality by extending **anInterface**. Sometimes, additional responsibilities (e.g. the ability to transform model elements) need to be

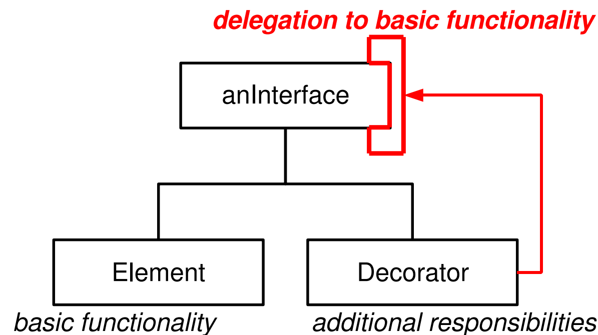

Figure 14: The basic interaction of the Decorator pattern

added without using inheritance, often because inheritance is already being used for another purpose. Then, a second class, **Decorator** can be introduced, which has two tasks:

1. **Additional behaviour.** The **Decorator** pattern introduces extra functionality to the class that it decorates. In our transformation tool, for instance, we use the Decorator pattern to add transformable behaviour to model elements.
2. **Standard behaviour.** The **Decorator** implements the same interface as (or a type-conforming subinterface of) the **Element** that it decorates (i.e. **anInterface**). Therefore, each **Decorator** also has to provide an implementation for the methods exported by that interface. Typically, the **Decorator** will delegate to the decorated **Element** for a concrete implementation of these methods. As such, code duplication between an **Element** and its **Decorator** is avoided.

So typically, a concrete decorator will implement additional behaviour and delegate to the decorated element (or a default implementation) for each operation published by the common interface (**anInterface**).

Application of the Decorator pattern in Pluto

Figure 15 shows how the Decorator pattern is used in the Pluto framework. The **TransFormableEntity** class implements the **ITransformableEntity** interface, which means that **TransformableEntity** also has to implement all the operations offered by **IEntity**. We now use the semantics of the Decorator pattern as described above:

- **Additional behaviour.** The **TransformableEntity** class implements all operations of the **ITransformableEntity** interface. This is done by inheriting from **TransformableERModelElementImpl**, which already implements a large part of these

operations. Because we use inheritance for reusing transformation logic, we cannot use inheritance for implementing the operations concerning dependency management, as published in the **IEntity** interface.

- **Delegation. TransformableEntity** contains an implementation of all operations defined in **IEntity**, but the implementation of these operations simply delegates to the **Entity** that is decorated by the **TransformableEntity**. **Entity** will in turn inherit a lot of common functionality from **ComposedERModelElementImpl**.

Thus, in our framework, decorators implement methods for executing transformations on model elements, and they delegate to the decorated model element in their implementation of those methods that are not concerned with model transformations.

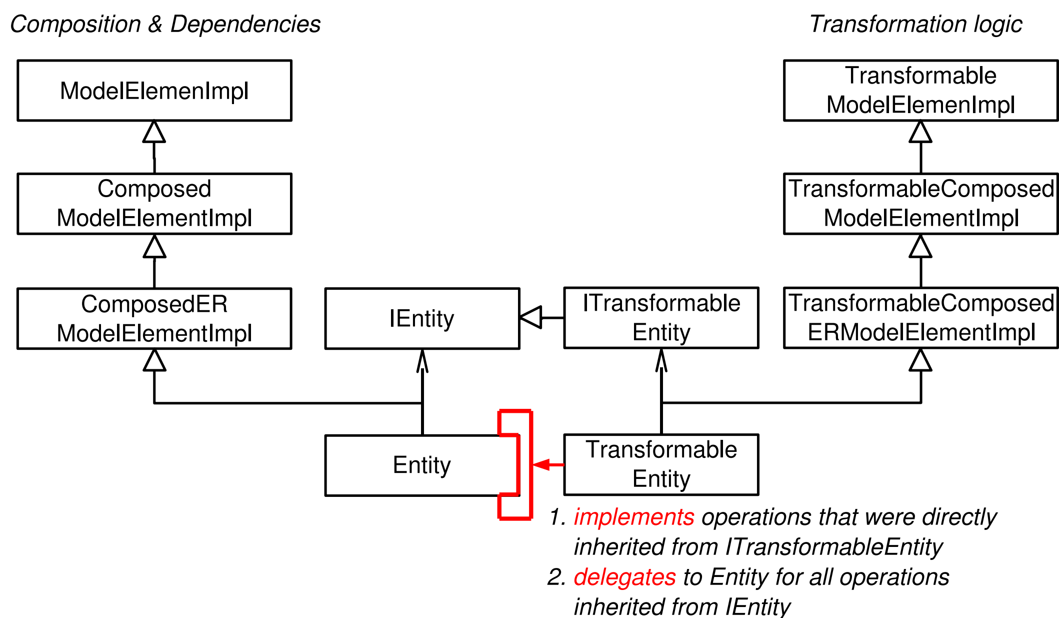

Figure 15: The Decorator pattern circumvents the lack of multiple inheritance in Java

9.3 Making Decorators Reusable in Cross-Model Transformations

Transforming Entity-Relation models into Relational Database models requires a completely different transformation strategy than transforming ER models into UML diagrams. This would require developers to implement separate decorators for each new target language to which they want to transform their models. Such decorators, however, introduce a lot of common functionality that is independent of the target language. Therefore, we have

isolated the *model-specific transformation logic* into separate objects according to the Strategy design pattern [9]. By isolating this model-specific transformation logic, decorators are made agnostic about the target language, thus increasing their reusability and applicability.

Strategies contain the technical details about transformations between two modeling languages. This design follows the Strategy pattern [9] and the concrete interactions of this pattern are shown in figure 16. An external actor (either the starter of a transformation or another decorator) sends a **(1)** **makeStrategy** request to the decorator of **SourceElement** and gives a reference to a **StrategyFactory** as an argument. The decorator then **(2)** asks the **StrategyFactory** for an appropriate **Strategy**, giving a reference to itself as an argument. This **StrategyFactory** now uses this reference to create **(3)** a new instance of a **Strategy**, which is returned **(4)** to the decorator of the **SourceElement**. After this procedure, the decorator is able to transform the **SourceElement** into a **TargetElement** by calling the **transform** operation of the concrete **Strategy**.

Obliviousness. Although extra functionality is added to the **SourceElement**, the latter remains *unaware* of any transformation logic because the association with its decorator is unidirectional. As such, we guarantee that metamodels are fully agnostic about transformation logic, which is one of the most important design goals of this project. The next section describes how these decorators and their accompanying strategies fit into the transformation algorithm that is provided by the Pluto framework.

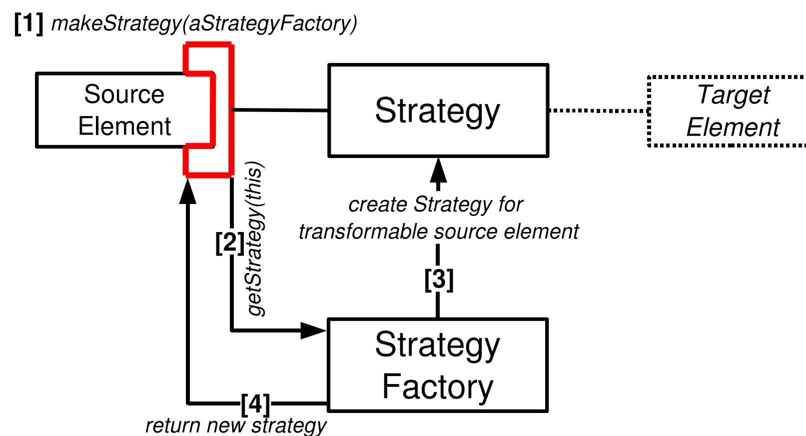

Figure 16: Strategies

10 Executing Model Transformations

After discussing how transformable behaviour can be attached to model elements, we now explain how Pluto provides a reusable transformation algorithm. This transformation algorithm will heavily rely on the *strategies* that were discussed in the previous section. First, we discuss the theoretic background of this transformation algorithm in section 10.1. Then, we show how it is implemented using the *transformation strategies* that were provided by developers (section 10.2). Finally, we give an example of a concrete model transformation in section 10.3.

10.1 Basic Transformation Algorithm

The basic idea underlying our transformation strategy is that *every model element* must be able to start a transformation. This is done in order to support local transformations. An example of a local transformation in an ER model is the addition of an attribute to an entity. Those transformations do not require the intervention of other entities and should therefore be executable at a local level. Obviously, localized transformations require each model element to know *how* and *when* such a transformation must be started. Luckily, these decisions are made based on (1) information that is local to the model element (its parent and dependees) and (2) on the following rules:

- **T1** Model elements can never directly transform other model elements; they can only transform themselves. It is only possible to start the transformation of another model element *indirectly*, as will be explained in section 10.1.1.
- **T2** Model elements can only transform themselves after all dependees and their parent (if any) have been transformed. This rule makes sure that the “container” has been transformed before the “contained elements” have been transformed.
- **T3** Model elements have to start the transformation of all their children and dependants after they have transformed themselves. This rule ensures that all the “contained elements” of a “container” will eventually be transformed.

These rules are independent of specific model transformations, so we have implemented them at the level of the Pluto framework. Next to these transformation rules, there are two preconditions on which the transformation algorithm relies:

- **V1** The model is *valid*, meaning that every model element obeys its invariants and that all dependee/dependant relations and parent/child dependencies are wired correctly.

V2 The model is *immutable* during the transformation. Changes made to the source model during the execution of a transformation are not reflected in the target model.

10.1.1 Conditional and actual transformations

A model element can never directly transform another model element, according to rule **T1**. This decision *localizes* the transformation strategy: before a model element can transform itself, it needs to ensure that all its *dependees* and its *parent* have been transformed (rule **T2**). Similarly, the model element has to make sure that the transformation request is propagated to all its *children* and *dependants* (rule **T3**). This information is local to the model element, making that element best suited to decide whether it is ready to be transformed. Therefore, we differentiate between two kinds of "transformations" —a public transformation operation, and a private one:

- **Conditional Transformation Requests (CT).** These requests are sent by external model elements to the model element that must be transformed. These requests are termed *conditional* because it is not guaranteed that the transformation will be executed *directly*. Indeed, the request may be preempted so as to conform to rule **T2**.

- **Actual Transformation (AT).** This is the active transformation, which can only be called by the model element itself. It does not check preemption constraints and it does not propagate calls to children or dependants. It just ensures that the model element is transformed properly. Concrete decorators implement this operation in two steps:

  1. **Create Transformation Strategy**. Strategies are created *online* when a transformable model element (i.e. a decorator) has to execute its transformation strategy. During the creation of such a *transformation strategy*, two validity checks are performed. First, **canHaveAsSourceElement** checks whether the source element is *assignable* to the strategy. It is not allowed, for instance, to attach strategies for transforming attributes to decorators of **Entity** instances. A second operation, **requires**, checks whether the target elements on which this strategy relies were already created by other strategies. Such target elements typically comprise the parents and dependees of the newly created target element (rule **T2**), and we need to wire these elements together so as to create a consistent target model. The **requires** test fails if a required target element has not been created, which typically points to an error in a concrete transformation strategy.

2. **Execute Transformation Strategy.** After the strategy has been created online, it can be executed. This causes the source model element to be transformed in a target model element according to the *execution plan* that is embedded in the strategy object. This step allows for *cross-model transformations* because different strategies can be attached to the decorator of a source model element (see section 9.3).

10.1.2 Transformation Algorithm

Our transformation algorithm follows directly from the rules about model transformations stated above (**T1–T3**). The algorithm is explained below and its subsequent steps are depicted in figure 17.

- **Step 1.** The transformation algorithm arrives **(1)** at **ModelElement** in either of these two ways: (1) a parent or a dependee has propagated the request to **ModelElement** as a result of applying rule **T3** or (2) a transformation request has arrived at one of the children or dependants of **ModelElement** and the transformation of that element was preempted because **ModelElement** is not yet transformed (as described in rule **T2**).

- **Step 2.** The model element first checks whether it has a *dependee* or a *parent* that has not yet executed its transformation (rule **T2**). If such an element (parent or dependee) exists, then the transformation of **ModelElement** is preempted and a conditional transformation request (**CT**) is sent to that element **(2)**. This causes the transformation request to be propagated *upwards* in the dependency lattice.
- **Step 3.** After the parent and all dependees of **ModelElement** have been transformed, then **ModelElement** itself is transformed **(3)**. This transformation is the actual transformation (**AT**), which lazily creates and executes a transformation strategy. This strategy manipulates the target model by creating a target element for **ModelElement**.
- **Step 4.** After **ModelElement** has transformed itself, it sends a conditional transformation request to all its children and dependants **(4)** so as to conform to transformation rule **T3**. This causes the transformation request to be propagated downwards the dependency lattice.

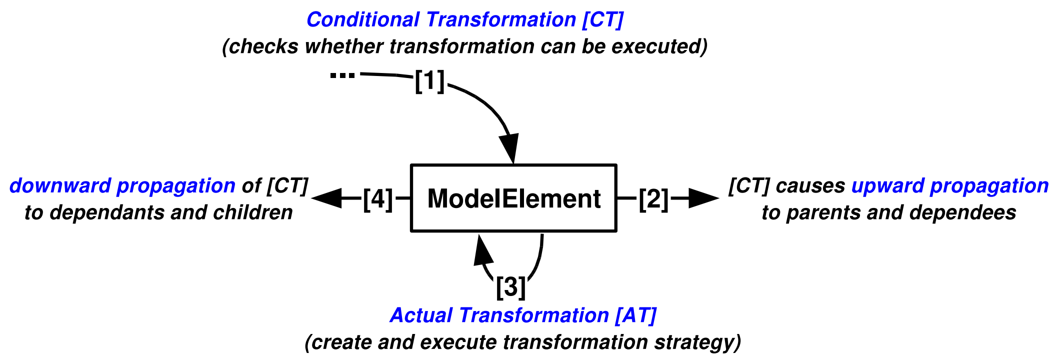

Figure 17: Overview of the Pluto transformation algorithm

10.2 Implementation of the Transformation Algorithm

This section focuses on the implementation of the transformation algorithm. It will become clear that the Pluto framework already provides most of the functionality of this algorithm, thus providing transformation implementors with a higher level of abstraction. The code that is specific for each model element is provided by the developer using the Template Method pattern. The overall structure of this design pattern is shown in the left side of figure 18. The black box represents the algorithm as a whole, which is typically hidden in a superclass, somewhere in the framework. Residing in a framework, the primary objective of such an algorithm is to be *generally reusable*. Sometimes, however, such algorithms comprise steps that cannot be implemented in a general way. Following the Template Method pattern, these steps are replaced by calls to *abstract methods*. These abstract methods thus serve as *templates* inside the algorithm, and they are represented as red boxes in figure 18. Programmers willing to reuse the algorithm, have to "fill in the templates" of that algorithm: they define a subclass of the framework class that implements the algorithm and provide an implementation for every abstract method, thus fine-tuning the behaviour of the algorithm without being confronted with the technical details of its execution strategy. As such, the responsibility of the framework user is reduced to implementing the red box in figure 18.

This design pattern is applied to our transformation algorithm as shown on the right side of figure 18:

- **Step 1. Transformation Preemption.** Transformation rule **T2** states that a model element cannot be transformed before all of its dependees and its parent are transformed. This is a general rule, so we have implemented it at the level of the Pluto framework.
- **Step 2. Model Element Transformation.** The transformation of a model element

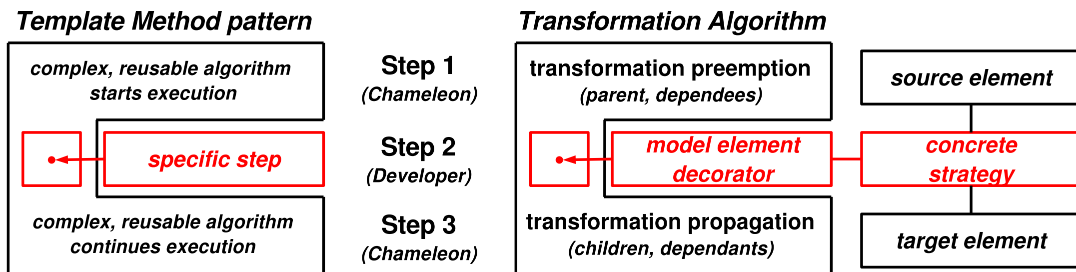

Figure 18: The Template Method pattern and its application in Pluto

is highly specific for two reasons. First, each model element is transformed in a different way. Entities from the ER model, for instance, are transformed differently than attributes from an ER model. Second, how a model element is transformed also depends on the target model. Transformations from ER to UML, for instance, differ significantly from transformations between ER and the Relational Database model. As Pluto is a general framework, these transformations cannot be implemented at this abstract level. Therefore, this step is to be provided by the implementor of the transformation. This is why the second step is represented by a red box (template method) in figure 18. This *template method* represents the actual transformation, **AT**, of the model element. During this phase, the decorator of the source element creates a new *transformation strategy*, which is executed so as to manipulate the target model.

- **Step 3. Transformation Propagation.** Transformation rule **T3** states that a ModelElement must propagate a transformation request to all its children and dependants. This is again a general rule, so we have decided to implement it at the level of the Pluto framework.

10.3 Illustration of the Transformation Algorithm

In order to illustrate our transformation strategy, we include a generic example and iterate through the steps of the recursive transformation algorithm. The example shows how a **ModelElement** is transformed and explains how this transformation affects the parent, children, dependees, and dependants of that **ModelElement**.

10.3.1 Setting

Consider the model depicted in figure 19, which is centered around a **ModelElement**. This element has a parent (**Parent**$_{ME}$), a dependee (**Dependee**$_{ME}$), a child (**Child**$_{ME}$), and a dependant (**Dependant**$_{ME}$). In the remainder of this example, we will transform the model based on a transformation request sent to **ModelElement**. First, this call propagates upwards to the parent and dependees of **ModelElement**, as explained in section 10.3.2. Then, the transformation request propagates downwards after **ModelElement** has been transformed, causing its children and dependants to be transformed, as explained in section 10.3.3.

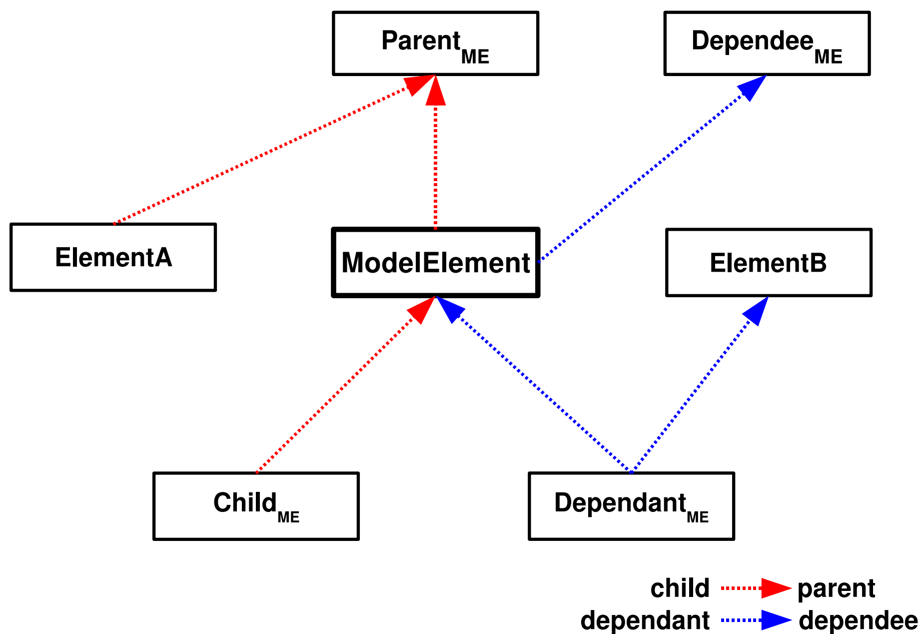

Figure 19: Example model to be transformed based on a call sent to **ModelElement**

10.3.2 Transformation Preemption in ModelElement

The upward propagation of the transformation request is depicted in figure 20, which shows a subset of the elements that were depicted in figure 19. **ModelElement** first checks whether its direct parents and dependees have already been transformed. After detecting that **Parent**$_{ME}$ is not transformed, **ModelElement** forwards a conditional transformation request **CT** to that parent **(1)**, causing the transformation algorithm to be started at

Parent$_{ME}$:

- In this example, **Parent**$_{ME}$ has no parents nor dependees, so there is no need to preempt the transformation algorithm for this model element and the second step of the transformation algorithm is immediately executed.

- **Parent**$_{ME}$ transforms itself using its actual transformation method **AT** **(2)**. This method is specific for each model element because different kinds of model elements tend to behave differently under transformations. Therefore, this **AT** operation is encapsulated into a transformation strategy object (to be provided by the developer).

- After **Parent**$_{ME}$ has finished its own transformation, the conditional transformation request **CT** is propagated to all the children and dependants of **Parent**$_{ME}$. These children are: **ElementA** **(3)** and **ModelElement** **(5)**.

Assume **Parent**$_{ME}$ first propagates the call to **ElementA** **(3)**. This model element checks whether its parent and dependees have been transformed, which is the case, so **ElementA** transforms itself **(4)** and propagates the call (**CT**) to its children and dependants. Both are unexistent, so the algorithm returns to **Parent**$_{ME}$, which was propagating conditional transformation requests (**CT**) to its children and dependants.

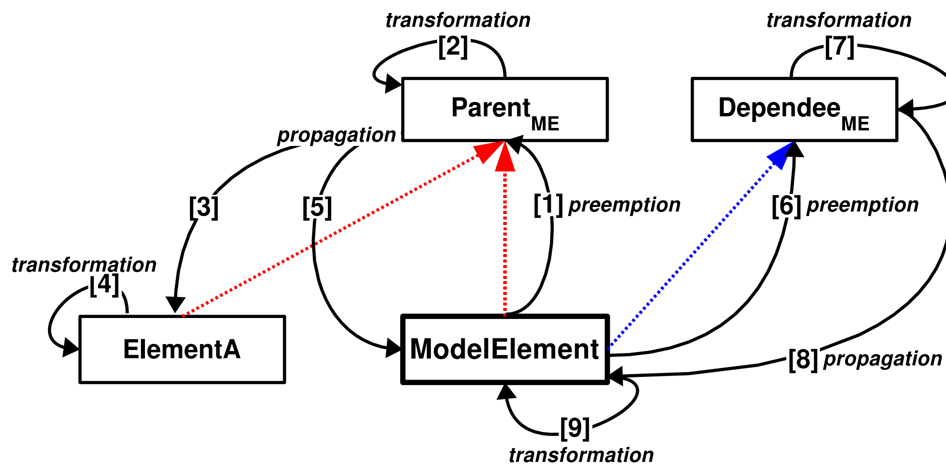

Figure 20: Transformation preemption in **ModelElement**.

Now, the second child, **ModelElement** receives the transformation request **(5)**. As in the beginning of this transformation, **ModelElement** checks whether its parents and dependees have already been transformed. This is true for **Parent**$_{ME}$, but it does not hold for all the dependees of **ModelElement** because **Dependee**$_{ME}$ has not yet been transformed. Therefore, the transformation of **ModelElement** is preempted **(6)** and control is given to

Dependee$_{ME}$. Since **Dependee**$_{ME}$ has no parents nor dependees, it can transform itself **(7)** and propagate the request to its children and dependants **(8)**. This causes **ModelElement** to be visited for the third time.

This time, however, during the execution of its conditional transformation method **CT**, **ModelElement** will notice that all its dependees and parents have been transformed, thus clearing the path for its own transformation, as implemented in its **AT** method **(9)**. This **AT** operation creates a strategy as discussed in section 9.3 and then executes this strategy.

10.3.3 Propagation from ModelElement to Children and Dependants

The behaviour of **ModelElement** after it has transformed itself is shown in figure 21. Since it has finished its own transformation, **ModelElement** can proceed to the third step of the transformation algorithm: it propagates the transformation request (**CT**) to its children (**Child**$_{ME}$) **(1)** and its dependants (**Dependant**$_{ME}$) **(3)**. After receiving this request, **Child**$_{ME}$ transforms without preemption **(2)** because all of its parents (**ModelElement**) and all of its dependees (which it does not have) have been transformed. **Dependant**$_{ME}$, on the other hand, has to wait **(4)** until **ElementB** has executed its transformation **(5)**. Then, **ElementB** propagates the call back to **Dependant**$_{ME}$ **(6)**, which causes the transformation of that element to be executed **(7)**.

Preemption and Propagation are not mutually exclusive phases. Note that preemption and downward propagation are not necessarily two distinct phases of a model transformation. In other words, preemption may still occur when a model element is propagating requests to its children and dependants. This is due to the hierarchical data structure of models, which is a *lattice* rather than a tree. An example is shown in figure 21. During the downward propagation of **ModelElement**, a conditional request is sent to **Dependant**$_{ME}$, but this request is preempted because the dependee of **Child**$_{ME}$, **ElementB** is not yet transformed. Thus, even though the algorithm is propagating calls to children, this does not mean that preemption will no longer occur during the transformation of the model.

Making the transformation order consistent. Because preemption and propagation do not comprise mutually exclusive phases in the algorithm, we disallow model elements to call the actual transformation method (**AT**) on their children and dependants. Such behaviour would bypass any upward propagations that are necessary before that child or dependant is transformed. For example, calling the actual transformation operation (**AT**) from **ModelElement** on **Dependant**$_{ME}$ would bypass the transformation of **ElementB**, thus yielding an inconsistent target model. Therefore, only model elements are allowed to invoke their own *actual transformation method* (**AT**), whereas parents and dependees

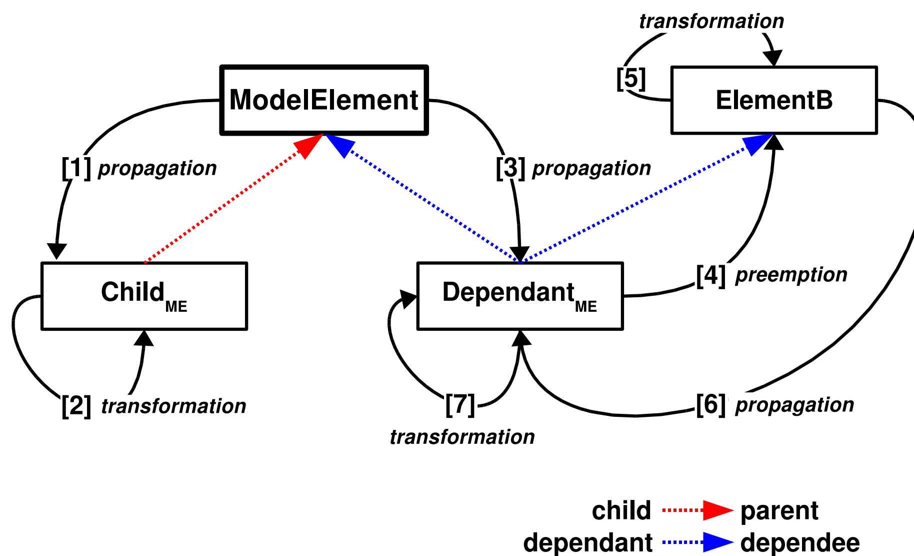

Figure 21: Downward propagation of the transformation request to the children and dependants of **ModelElement**

are forced to call the *conditional transformation request* (**CT**) on their children and dependants. The same holds in the reverse direction: children and dependants are allowed to send conditional transformation requests to their parents and dependees, but they may never directly invoke the actual transformation strategy.

10.4 Discussion

A summary of the design of the transformation algorithm is shown in figure 22. In this section, we provide a discussion on the design patterns that underly this design.

The Decorator pattern. This design pattern is primarily used to circumvent the lack of multiple inheritance in Java. The main advantage of using the Decorator pattern is that metamodels are not polluted with transformation logic. Indeed, code for transforming model elements is abstracted into reusable decorators and these decorators are called by the transformation algorithm when the decorated source model element must be transformed. Although the decorator pattern is a suitable candidate for ensuring metamodel obliviousness, multiple inheritance would have been far better because the decorator still needs to delegate to the operations exported by the source model element that it decorates. These delegations are trivial to write, but they distract the focus from the real business logic of the

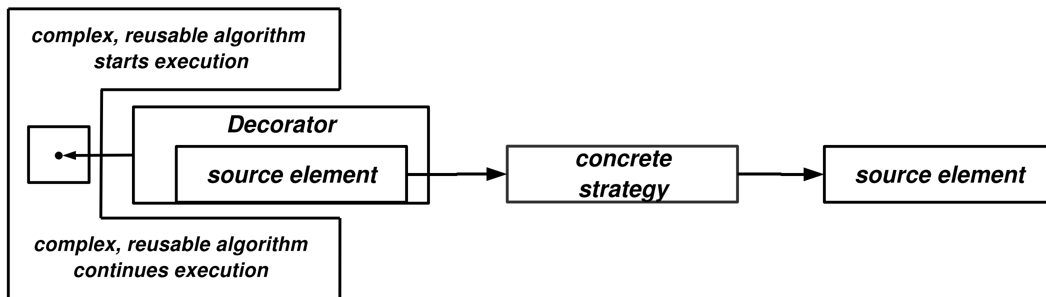

Figure 22: Summary of the transformation algorithm

decorator.

The Strategy pattern. The strategy pattern is introduced so as to make the decorators independent of the target modeling language to which a source model is transformed. Model transformations from ER to RDB require other transformation strategy objects than transformations between ER and UML, but the decorators of the source model can be reused. Another advantage of the encapsulation of this transformation logic is that it becomes easier to compile future programming languages defined for model transformations into concrete strategies. Indeed, such compilers do not have to take into account the logic that is provided by the model element decorators. They only have to work with concrete strategy classes.

The Template Method pattern. The use of the Template Method pattern has two advantages. First, developers are given a higher level of abstraction because they are able to reason about transformations at a local level. They no longer have to consider the transformation as a whole and they can concentrate on the transformation of specific model elements instead. Second, by hardwiring the first and the last step of the transformation algorithm in the Pluto framework, the Template Method pattern ensures that propagation of transformation requests is done consistently and at the right time, thus reducing the probability to write inconsistent or incorrect transformations at the level of a subclass.

11 Related Work

In [21] and [24], the applicability of Action Semantics (AS) [15] to model transformations is studied. The authors conclude that AS can be used for transforming between UML models, thus allowing for iterative refinement of UML designs. Another advantage of using AS is that design patterns can be encoded as a sequence of transformation steps, thus

allowing to *refactor* designs [7]. One shortcoming is that cross-model transformations are not supported because AS is irreversibly linked to the Unified Modeling Language.

YATL [16] and Atlas [4, 3] are languages for defining model transformations. They combine declarative concepts for querying the source model with imperative constructs for executing the transformation itself. By relying on the Meta Object Facility [2], these languages support pluggable metamodels, but they do not provide reusable concepts for *defining* those metamodels. Furthermore, it is not clear how these transformation languages can be applied to concrete instances of such newly introduced metamodels.

MTRANS [17, 18] is a model transformation framework that provides both a development environment and a language to define model transformations. This language is defined as an abstraction above XSLT and, therefore, the transformation architecture of MTRANS is strongly influenced by the XSLT specification [1]. One major drawback of this dependency is that many-to-one transformations are not supported because XSLT inherently relies on one-to-one mappings between source and target elements. Chameleon, on the other hand, is independent of any transformation language and transparently manages one-to-many and many-to-one dependencies for model transformations.

In [13], a model transformation approach based on *meta templates* is proposed. These templates are reusable XML-esque descriptions of transformation rules to be executed by a transformation tool. One shortcoming when compared to Chameleon is that the language provided in this approach is not checked for type-safety and model consistency, thus allowing to create invalid model dependencies in the target model. In Chameleon, on the other hand, such model anomalies are disallowed by the model checker, which is available at the abstract framework level and can hence be reused and specialized by concrete metamodels.

UMLX [26] and VMT [20] are graphical transformation languages for MDA, primarily developed in an attempt to increase user-friendliness of model transformation languages. The major advantage of UMLX and VMT is their expressiveness, given their limited amount of graphical modeling constructs. One problem, however, is that these transformation languages are limited to transforming UML models. As the UMLX compiler is able to compile transformations into Java code, however, it should be possible to attach this generated code to our transformation strategies. Indeed, our model transformation framework can also be used to *transform between programming languages*. Concrete examples of such transformations can be found in the work of [23] and [6]. By integrating Chameleon's support for language transformations with the VMT compiler, we integrate a visual transformation language with *cross-model transformability*, thus solving the problems of the original VMT proposal, which relies on a hardwired metamodel.

A detailed analysis of the application of MDA to workflow-supported model transformations in product line systems was conducted in [11]. Their MT-Flow tool allows to transform between different workflow modeling languages by relying on pluggable metamodels. It is, however, impossible to transform *beyond* workflow languages and the tool lacks support for *customizing* or *extending* metamodels.

Finally, a large number of transformation languages have been proposed, for example, Converge [22] and the work of Kuznetsov [12]. These languages are typically compiled and executed on a transformation tool that relies on a hardwired metamodel, thus disallowing *pluggable metamodels*. We are investigating how these transformation languages can be compiled to our transformation strategy objects, which is beneficial for both paradigms: (1) the transformation language can be used for cross-model transformations and (2) the developer is freed from having to program strategies.

12 Future Work

During the design of the current version of our transformation tool, we have focused on design goals related to basic model transformations. After having completely implemented and tested this framework in the context of the ER and RDB metamodel, we see a number of future directions, which are summarized in this section.

- **Rollback.** There are a number of scenarios in which transformations must be rolled back. Entirely undoing a transformation may be needed, for instance, when earlier developers want to backtrack their designs to an earlier version. Also, partially rollbacking a model transformation may be necessary when concrete models contain *specific dependencies* that are not modeled at the level of the Pluto framework, which can only enforce general dependencies. Therefore, we plan on implementing an *undo mechanism*. This mechanism requires developers to provide an **undoTransform()** method next to the **transform()** operation in each transformation strategy.
- **Grouping.** Complex transformations require functionality for transforming N source elements to 1 target element and vice versa. This creates a need for a *grouping mechanism* that transforms an entire group of source elements to one target element. Currently, Pluto does not provide full-blown support for group such source model elements.
- **Storing.** Next to executing and undoing transformations, we also need a mechanism for *persisting transformations*. This can be done at two levels. First, concrete, model-specific transformations can be stored. A simple example is the transformation of all

entities of which the name starts with an A to entities with a name starting with a B. Second, abstract, model-independent transformations should be persistent. An example of such an abstract transformation is the transformation of entities to tables. Storing these entities in a machine-readable format eventually leads to (1) a reusable store of executable transformations and (2) a transformation language with language concepts that are based on these stored transformation definitions.

13 Conclusion

Modeling tools often rely on hardwired, proprietary metamodels, as such obstructing *cross-model transformations* and metamodel reuse. This leads to inconsistent designs scattered over a variety of modeling tools. In this text, we have presented the design and the implementation of the Pluto framework. This framework provides reusable concepts for *metamodeling* and for *model transformations*. Pluto eases the definition of new metamodels by providing reusable concepts for *dependency management* and *model composition*. Furthermore, Pluto enables cross-model transformations by deferring model-specific transformation logic to *strategies* containing localized execution plans. These strategies are referred to by *decorators* so as to keep the metamodel agnostic about details about model transformations.

The decoupling between *metamodel-independent* constructs offered by Pluto and *model-specific* concepts provided by *developers* decreases the development time of new metamodels and increases their consistency because modellers can focus on metamodel-specific concepts, at the same time inheriting common modeling functionality from the Pluto framework.

References

[1] *XSL Transformations Version 1.0, http://www.w3.org/TR/xslt*. 1999.

[2] *Meta Object Facility Core Specification 2.0 http://www.omg.org/technology/documents/formal/MOF_Core.htm*. 2006.

[3] Jean Bezivin, Frederique Joualt, and Patrick Valduriez. On the Need for Megamodels. In *Automated Software Engineering. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2004.

[4] Jean Bezivin, Patrick Valduriez, and Frederique Joualt. The ATLAS Transformation Language, http://www.sciences.univ-nantes.fr/lina/atl/, 2004.

[5] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA 2003, Workshop on Generative Techniques in the Context Of Model-Driven Architecture*, 2003.

[6] Sven De Labey, Marko van Dooren, and Eric Steegmans. ServiceJ. A Java Extension for Web Service Interactions. In *Proceedings of the Fifth IEEE International Conference on Web Services (ICWS'07)*, Salt Lake City, Utah, July 2007.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[8] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. 2003.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[10] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The Missing Link of MDA. In *A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, Proc. Graph Transformation - First International Conference, ICGT 2002*, 2002.

[11] Jernej Kovse and Theo Harder. MT-Flow – An Environment for Workflow-Supported Model Transformations in MDA. In *Advanced Information Systems Engineering*, 2004.

[12] Mikhail Kuznetsov. Automated Model Transformation in MDA. In *Colloquium on Database and Information Systems*, 2005.

[13] Hongming Liu, Lizhang Qin, Xiaoping Jia, and Adam Steele. Model Transformation Based on Meta Templates. In *International Conference on Software Engineering Research and Practice*, 2006.

[14] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[15] Object Management Group. *ptc/02-09-02: UML 1.5 – Action Semantics*. 2002.

[16] Octavian Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*. University of Twente, the Nederlands, January 2004.

[17] Mikael Peltier, Jean Bezivin, and Gabriel Guillaume. MTRANS: A General framework based on XSLT for model transformations. In *WTUMLO1; Proceedings of the workshop on transformations in UML*, 2001.

[18] Mikael Peltier, Francois Ziserman, and Jean Bezivin. On levels of Model Transformations. In *XML Europe 2000, pages 1–17*, 2000.

[19] Shane Sendall and Wojtek Kozaczynski. Model Transformation — the heart and soul of model-driven software development. In *IEEE Software, September/October 2003 (Vol. 20, No. 5)*, 2003.

[20] Shane Sendall, Gilles Perrouin, Nicolas Guelfi, and Olivier Biberstein. Supporting Model-To-Model Transformations: the VMT Approach. In *Technical Report TR-CTIT-03-27, Twente*, 2003.

[21] Gerson Sunye, Alain Le Guennec, and et al. Using UML Action Semantics for Model Execution and Transformation. In *The 13th IC on Advanced Information Systems Engineering*, 2002.

[22] L. Tratt and T. Clark. Model transformations in Converge. In *Workshop in Software Model Engineering (WiSME)*, 2003.

[23] Marko van Dooren and Eric Steegmans. Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions using Anchored Exception Declarations. In *International Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005.

[24] Varro, D. and Pataricza, A. UML Action Semantics for Model Transformation Systems. In *Periodica Politechnica*, 2003.

[25] Jos Warmer and Anneke Kleppe. *The Object Constraint Language – Getting your Models Ready for MDA*. 2003.

[26] Willink, E. D. UMLX : A graphical transformation language for MDA. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.