A FRAMEWORK FOR EXECUTING CROSS-MODEL TRANSFORMATIONS BASED ON PLUGGABLE METAMODELS

Geert Delanote, Sven De Labey, Koen Vanderkimpen and Eric Steegmans
K. U. Leuven, Department of Computer Science, 200A Celestijnenlaan, B-3000 Leuven, Belgium

Keywords: Framework, metamodeling, model transformations.

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.

1 INTRODUCTION

The design of complex enterprise applications imposes new challenges on software developers as well as on tools supporting model-driven software design. Such complex designs are typically created using different modeling languages, where each language focuses on a specific aspect of the enterprise application. Aligning software with business requirements, for instance, is typically done using workflow modeling languages such as the Business Process Execution Language (BPEL) (BPEL4WS, 2003). The technical realization of each identified BPEL process is then modelled using UML interaction diagrams, whereas the back-end database persisting the state of that BPEL process is designed using Entity-Relationship (ER) or Relational (RDB) database models. The heterogeneity of these modeling languages imposes new challenges on modeling tools. For example, Platform Independent Models crafted in earlier stages of the design process must be transformable to models written in different modeling languages, thus creating a need for *cross-model transformations* (Frankel, 2003). Also, general design decisions to be respected by the whole project must be communicated to models written in different modeling languages, as such creating a need for *cross-model communication* in order to guarantee *cross-model consistency* (Rensink, 2005).

Current tool support can hardly cope with the increased complexity introduced by heterogeneous modeling languages as the majority of modeling tools rely on *hardwired, vendor-specific metamodels*. By hardwiring their metamodels, such tools disable transformations to other modeling languages. Furthermore, by relying on *proprietary* metamodels, transformation tools obstruct metamodel *reuse* in other modeling tools because those competing tools rely on ad hoc metamodels that are in turn proprietary and hence incompatible.

Being convinced that a lack of support for cross-model transformations and vendor lock-in are serious limitations of today's modeling tools, we have designed and implemented a *transformation framework* with support for *pluggable metamodels*. By making metamodels interchangeable, this tool allows mod-

Delanote G., De Labey S., Vanderkimpen K. and Steegmans E. (2007).
A FRAMEWORK FOR EXECUTING CROSS-MODEL TRANSFORMATIONS BASED ON PLUGGABLE METAMODELS.
In *Proceedings of the Second International Conference on Software and Data Technologies*, pages 315-325
DOI: 10.5220/0001348803150325

ellers to introduce a metamodel for their own modeling language and it provides a basis for executing horizontal, *cross-model transformations*.

In this paper, we focus on the *design* of *Pluto*, a framework providing reusable concepts for (1) building concrete metamodels and (2) for transforming concrete models between different modeling languages. The remainder of this text is structured as follows. Section 2 elaborates on current tool support for model transformations and discusses a number of limitations, leading to a list of design goals, as described in Section 3. Section 4 introduces our framework, Pluto, and Section 5 shows how Pluto eases the implementation of new metamodels. Section 6 shows how Pluto models can be transformed to target models written in other modeling languages. Finally, Section 7 discusses related work and Section 8 concludes.

2 MOTIVATION

Current tools for designing and transforming models hardly support the functionality required for developing complex software systems. We have identified a number of disadvantages that have led to the design goals enumerated in Section 3:

- **Limited Applicability.** Transformation tools are often shipped containing a *hardwired metamodel* of a *single modeling language* so as to (vertically) transform models *within that same language*. Although complex models may be realized using such tools, it is impossible to design models in any other modeling language than the one supported. This forces modellers to use a different tool for each modeling language they are willing to use. Threatened by the risk to scatter their designs over incompatible tools, developers are tempted to stick with a single, general modeling language such as UML, even though specialized modeling languages are often better suited for modeling specific parts of a software system.
- **Limited Interoperability.** Next to being hardwired in modeling tools, embedded metamodels often contain *proprietary* constructs, for example, to increase the performance of the transformation tool in which they are embedded. Although economically feasible for the tool supplier, who achieves a vendor lock-in, such proprietary constructs are awkward for the *end users* of that tool because it becomes very hard to reuse their designs in other modeling applications. Indeed, the latter will not be able to understand the format of the *proprietary metamodel* in which the original version of the model was defined.
- **Limited Extensibility.** Modeling tools often provide a mechanism to define *extensions* for modeling languages. The Unified Modeling Language, for instance, introduces *UML profiles* and *stereotypes* to allow developers to extend UML with domain-specific modeling concepts (OMG UML Specification 1.5, 2003). Transformation tools may provide similar extension mechanisms to let developers extend the modeling language on which that tool operates. The *expressive power* of such mechanisms, however, is much weaker than that of a *pluggable metamodel system* because the semantics of metamodel-specific extension mechanisms are irreversibly linked to the semantics of their *base metamodel*, thus obstructing the introduction of new modeling concepts that are incompatible with that base metamodel.
- **Limited Support for Transformations.** Hard-coding a metamodel inside the transformation tool disables transformations *between* modeling languages. This makes it impossible, for instance, to transform a series of BPEL processes into a UML interaction diagram. Indeed, tools used for creating such models will lack the ability to communicate design decisions to each other, thus jeopardizing the consistency of the application being designed.
- **Limited Reusability.** Although specialized tools exist that are able to transform between multiple modeling languages, the set of supported languages is typically predefined and therefore not extensible. Also, the logic for executing such transformations is often too much tailored to the base application, hence obstructing the reuse of transformation algorithms in other tools.

3 DESIGN GOALS

The main objective of our research project is to build a *metamodel-independent* transformation tool that supports transformations between models designed in *different modeling languages*. This gives rise to a number of design goals:

- **Pluggable Metamodel System.** Developers must be able to define their own metamodels and feed them to the transformation tool, as such increasing the applicability of the latter. For example, if our transformation tool is running on a UML metamodel, it must be able to accept an ER metamodel or an RDB metamodel and then allow modellers to design their applications using the UML, ER and RDB modeling languages. Next to adding

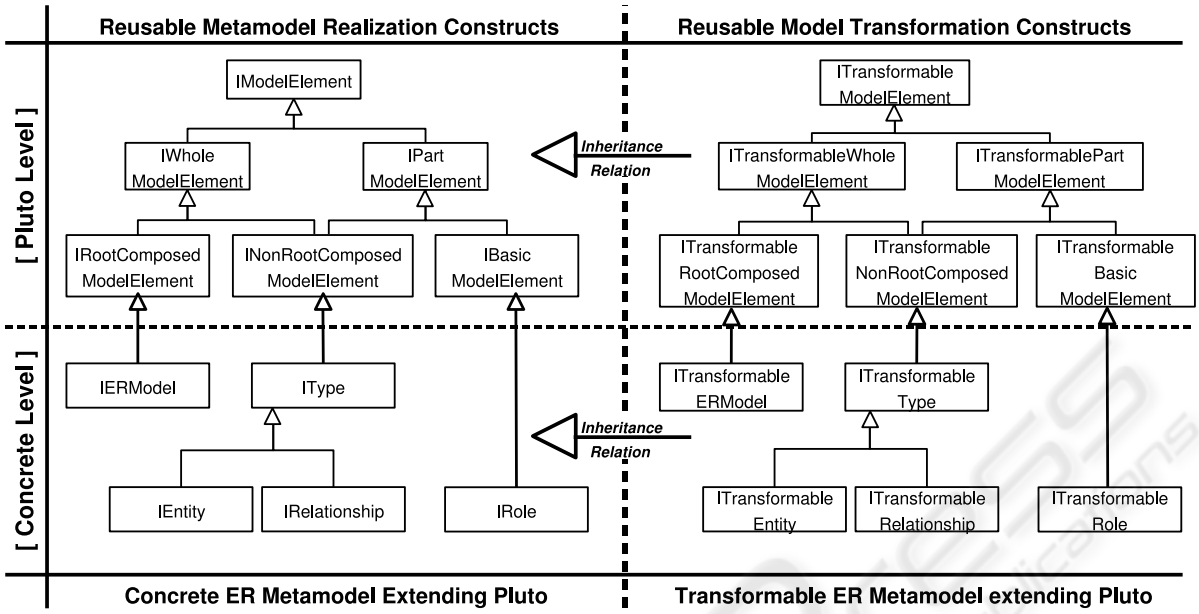

Figure 1: Overview of the Pluto framework for metamodeling.

metamodels of existing modeling languages, developers must also be able to define and insert *custom metamodels* of domain-specific modeling languages or language extensions that they have defined themselves.

- **Cross-model Transformations.** Next to transforming between models that share a common metamodel, our tool must support transformations between *different modeling languages* (Mens et al., 2005). This requires our tool to be able to work with multiple metamodels *simultaneously*. When fed with the ER and the UML metamodel, for instance, it must be possible to transform a UML class diagram into an ER schema, and vice versa.
- **Reusability.** Metamodels have a wide-spread applicability beyond the domain of model transformations as they can be used, for instance, in model verifiers or code generators. Therefore, the metamodels on which our tool relies must be *reusable* in these domains without modification: we must avoid polluting metamodels with dependencies on components that are specific to model transformation logic.
- **Obliviousness.** In order to allow the transformation tool and the metamodels to evolve independently, a certain amount of obliviousness is needed in both directions. On one hand, metamodels should be entirely independent of the transformation tool (as noted in "*reusability*"); on the other hand, the transformation tool should not depend on the technical details of a single hard-wired metamodel (as noted in "*cross-model transformations*"). Thus, we need to define a *common contract* on which the transformation tool relies when a metamodel is fed to it.

Realization. Given these design goals, we have developed *Pluto*, a framework with reusable concepts for metamodels and model transformations. Section 4 outlines this framework. Its two main functions, *metamodel realization* and *model transformation*, are discussed in Sections 5 and 6, respectively.

Traceability. The need for horizontal, cross-model transformations in turn creates a need for *cross-model communication* so as to guarantee the consistency of the design. Therefore, transformations must be *traceable*, meaning that our tool must be able to link target elements to the source element(s) that triggered the creation of the former. This also means that changes to a target model must be communicated back to the source model from which that target model was created.

4 THE PLUTO FRAMEWORK – OVERVIEW

Pluto is a Java framework providing abstract modeling constructs to be reused by concrete metamodels. It is therefore typically layered on top of *concrete* metamodels as shown in Figure 1. The upper part depicts a set of Pluto constructs providing general func-

tionality for metamodels. The lower part of Figure 1 shows how interfaces of the ER metamodel are defined as *extensions* of Pluto interfaces. Concrete realizations of these interfaces represent elements that occur in ER models. Such concrete classes implement these ER-specific interfaces by inheriting general functionality from abstract Pluto classes and by filling in ER-specific behaviour where necessary.

It is important to note that Pluto is in itself *not a metamodel of metamodels* and neither is it a *concrete metamodel*. Rather, Pluto should be seen as an *abstract metamodel* from which concrete metamodels inherit common functionality. Therefore, in the Meta Object Facility layering structure (The Meta Object Facility Core Specification 2.0, 2006), Pluto is situated at the meta-level (M2) rather than the meta-meta-level (M3).
The remainder of this text focuses on two key functions of Pluto.

- **Metamodeling.** Section 5 focuses on the *left part* of Figure 1 and shows how Pluto offers reusable concepts for *composition* and *dependency management*. To illustrate the wide applicability of this transformation framework, we use the Entity Relationship (ER) metamodel and the Relational Database (RDB) metamodel as examples instead of the UML metamodel.
- **Transforming Models.** Section 6 reviews the *right part* of Figure 1 and shows how Pluto transforms models between different modeling languages. As an example, we show how the ER metamodel can be extended so as to transform models from the ER modeling language to the RDB language, and vice versa.

5 DESIGNING CONCRETE METAMODELS AS PLUTO EXTENSIONS

Pluto offers constructs for building *consistent, hierarchical composition structures* (Section 5.1) and provides a *reusable dependency management system* (Section 5.2) that can be reused by developers of concrete metamodels. Both constructs provide functionality that is paramount for Pluto's transformation logic (Section 6).

5.1 Reusable Composition Concepts

The existence of nested *whole-part relations* typically causes models to be arranged into *hierarchies*. The `ERModel` instance in Figure 1, for example, refers to a number of `Relationship` instances, and each `Relationship` in turn contains a number of relationship ends (represented by instances of `Role`). Similar composite structures occur frequently in the majority of available modeling languages. Without having reusable constructions for managing model composition, however, implementors of metamodels have to implement similar structures over and over again. We avoid this repetitive and error-prone work by integrating reusable functionality for managing hierarchies at the level of Pluto. This is achieved by structuring its top level classes according to the *Composite* design pattern (Gamma et al., 1994). This composite structure is shown in Figure 2, where instances of `WholeModelElement` refer to instances of `PartModelElement`. The basic version of the Composite pattern is further extended by the introduction of three *subinterfaces* that differentiate between the root node, the intermediate nodes, and the leaves of a model tree:

- `RootComposedModelElement` is used for elements that are not contained in other elements, such as the *root node* of a model. In the ER metamodel (see Fig. 2), for instance, this interface is extended by the `ERModel` interface because ER models cannot be contained in other model elements.
- `BasicModelElement` is used to represent those model elements that are contained in other model elements without containing elements themselves. They represent the *leaves* of a model tree. An example of such a basic model element is the relationship end of an `Entity`, as represented by `Role` in Figure 2.
- `NonRootComposedModelElement` inherits its behaviour from both `WholeModelElement` and `PartModelElement`, meaning that it contains model elements while being contained in another model element. Concrete realizations of this interface represent *internal nodes*. The `Relationship` interface, for instance, is an internal node because it is contained in an `ERModel` (see Figure 2) while containing a set of `Role` instances.

Evaluation. By augmenting the semantics of model composition, we allow metamodel developers to *reuse* Pluto's consistency and validity checks, as such decreasing the odds for introducing *structural integrity violations* (e.g. circular dependencies or detached model elements) during the construction of a new metamodel.

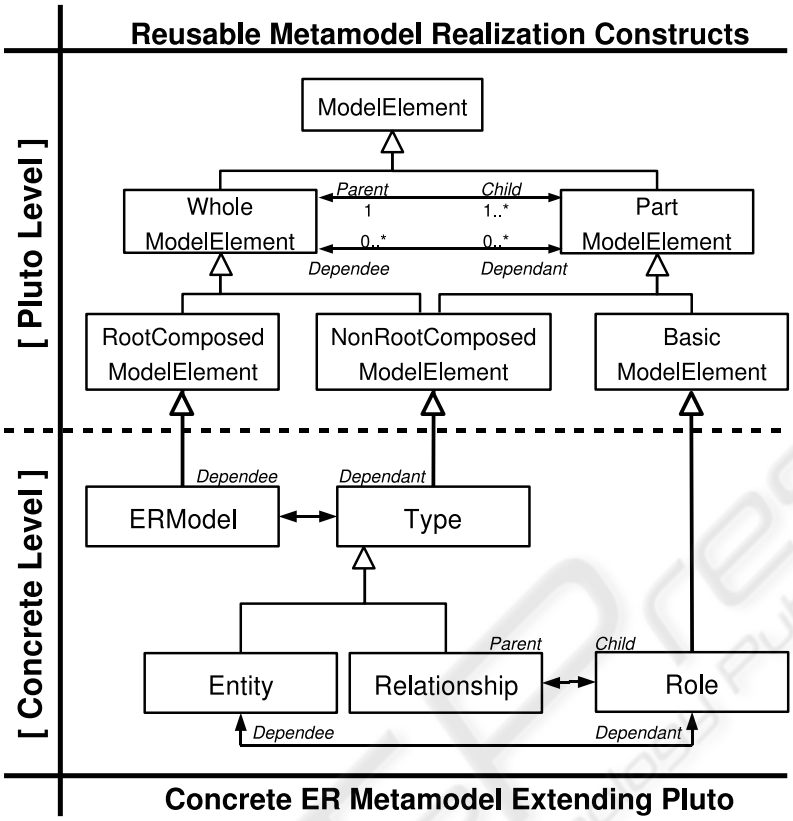

Figure 2: Parent/child and dependee/dependant relations in Pluto.

5.2 Reusable Dependency Management

The *whole/part* relation of Section 5.1 is too general to cover different kinds of dependencies between model elements. Therefore, this section further decomposes that relation into *two specialized dependency relations*. First, the *parent/child* relation is used to specify *interdependencies*. Second, the *dependee/dependant* relation is used to specify *unidirectional dependencies* between different kinds of model elements.

The Parent/Child Relation. This is a hierarchical one-to-many relation used for expressing interdependencies between different kinds of model elements. The following properties characterize the parent/child relation:

- **One-to-Many.** A parent can have multiple children, but a child only has one parent. For example, the realization of the ER metamodel specifies that a `Relationship` (parent) contains multiple relationship ends (i.e. instances of `Role`), but a `Role` (child) can only belong to one `Relationship`.
- **Interdependence.** The parent/child relation is used to express the existence of *bidirectional dependencies* between different kinds of model elements. This means that parents and their children influence each other:
  - *Children depend on their parent.* Children cannot survive the removal of their parent. In the ER metamodel, for instance, this means that instances of `Role` cannot survive the removal of the `Relationship` to which they belong. This is exactly what the ER metamodel must enforce because a role does not make sense when it is not participating in any relation.
  - *Parent depends on children.* In parent/child relations, the parent also depends on its children. A `Relationship`, for instance, is invalid without instances of `Role` being attached to it; such a configuration would specify the existence of a relationship without participants and this is not sensible.
- **Transitive.** Given three model elements α, β, and γ, such that α is a parent of β and β is a parent of γ, then α is an indirect parent (or *ancestor*) of γ.
- **Symmetric.** A model element α is the parent of

β iff β is a child of α. Combined with *transitivity*, we get that α is an *ancestor* of β iff β is a *descendant* of α.

- **Non-reflexive and Acyclic.** The data structure induced by the parent/child relation is *acyclic*. No model element can be its own parent and a child cannot be the parent of one of its ancestors.
- **Tree structure.** The non-reflexive, transitive closure of the parent/child relationship induces an *acyclic tree structure* on a set of model elements.

The Dependee/Dependant Relation. This relation models 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.
- **Unidirectional Dependency Relation.** Unlike the parent/child relation, where a parent depends on its children, dependees are *independent* of their dependants. In the ER metamodel of Figure 2, for instance, an `Entity` is a dependee of its `Role` instances (and not a parent) because the ER specification does not require an `Entity` to participate in a `Relationship` to be valid. The reverse, however, remains unchanged: the deletion of an `Entity` is cascaded to its `Relationship` instances in order to ensure model consistency.
- **Transitive.** Similar to the parent/child relation, dependee/dependant is a transitive relation. In Figure 2, for instance, `ERModel` is a dependee of `Entity`, which is in turn a dependee of `Role`. Therefore, `ERModel` is an *indirect dependee* of `Role` whereas `Role` is an *indirect dependant* of `ERModel`.
- **Non-reflexive, Symmetric, 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 induces an acyclic *lattice* structure on a set of related model elements, rather than a *tree*.

Reusable Dependency Management. It is clear that mathematical properties such as *symmetry*, *transitivity* and the abscence of *cyclic dependencies* require rigorous specifications for managing the dependencies between different kinds of model elements in trees and lattices. Without a reusable infrastructure for dependency management, metamodel implementors would have to implement these relations over and over again, thus increasing the possibility of introducing inconsistencies in the metamodel. By integrating dependency management at the level of Pluto, however, concrete metamodels extending this framework inherit (1) fine-grained consistency guarantees and (2) algorithms that automatically manage dependencies based on lifecycle changes of model elements, thus easing the implementation of new metamodels.

6 MODEL TRANSFORMATIONS

Next to offering reusable constructs for building metamodels, Pluto incorporates concepts for *transforming* models defined as concrete instances of those metamodels. These concepts are shown in the right part of Figure 1. The basic idea of model transformations in Pluto is shown in Figure 3, which shows how transformable behaviour is attached to model elements using the *Decorator pattern* (Gamma et al., 1994). These classes decorate model elements with transformable behaviour by pointing to a *strategy* containing an *execution plan* for creating a target element. Developers willing to transform some source model to a UML class diagram, for instance, attach UML-specific transformation strategies to that decorator, whereas database modellers will select RDB- or ER-specific strategies. This section first explores how transformation strategies are added to metamodels via decorators (Section 6.1) and then shows how these transformation strategies interact with Pluto's generic transformation algorithm (Section 6.2).

6.1 Decorating Model Elements with Transformation Strategies

The upper right part of Figure 1 shows the key interfaces exported by Pluto that attach transformable behaviour to model elements. There is a one-to-one mapping between transformation-specific interfaces (e.g. `TransformableModelElement`) and basic interfaces for metamodels (e.g. `ModelElement`) as shown in Figure 4. There is also a *default implementation* for each transformation-specific interface to be reused by decorators of concrete metamodels. This is also shown in Figure 4, where a concrete decorator, `TransformableEntity`, inherits from a Pluto class in order to decorate `Entity`. Such concrete decorators must be created only once for each metamodel element because they are *fully agnostic* about technical transformation details. They only serve as a "bridge" between model elements and their *execution plan*. These execution plans provide mappings between source elements and target elements,

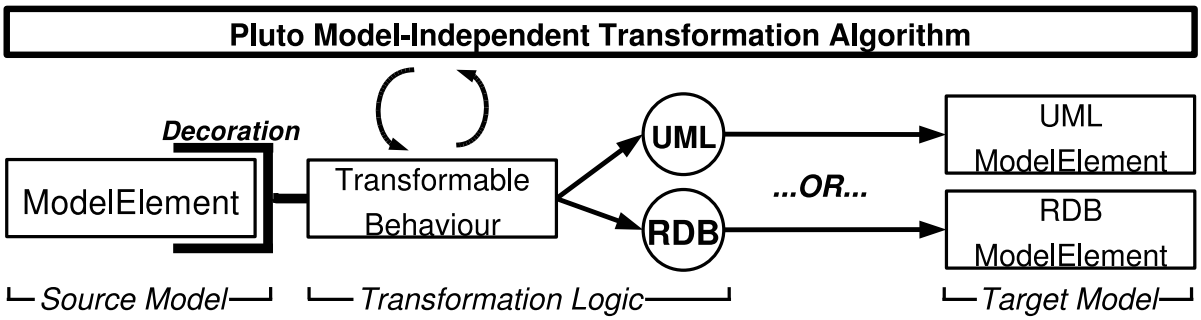

Figure 3: Transformation Overview.

so they are encapsulated into *transformation strategies*, shown as *circles* in Figure 4.

Transformation Strategies. The main responsibility of a strategy is to *encapsulate an execution plan* for the transformation of a source element into the corresponding model construct of a target language. Figure 4, for instance, shows two different transformation strategies that can be attached to a `TransformableEntity` decorator. Each strategy contains a different execution plan because they both transform instances of `Entity` to a different target modeling language –UML and RDB, respectively.

Each strategy publishes an operation `transform()`, which is called by Pluto through the decorator to which the strategy is attached. The execution of such a transformation strategy only has a *local effect*; it transforms a single model element, independent of its surrounding elements. Any dependencies with related model elements are handled by the dependency relations that were integrated in Pluto, as discussed in Section 5.2. Such localized transformations make the implementation of strategies straightforward, as such making them eligible for *code generation*. Although not a target of this research project, it should be manageable to add an extra layer above Pluto that converts model transformation languages into strategies that can be executed by our transformation algorithm.

6.2 Pluto's Generic Algorithm for Transforming Model Elements

The previous Section explained how transformation strategies are attached to decorators in order to make model elements transformable. This Section shows how Pluto's generic algorithm interacts with these strategies in order to transform model elements. The basic philosophy of this algorithm is to exonerate the developer from taking care of all the technical "middleware" concerns of model transformations. Such tasks include determining an execution order, managing cross-model consistency, ensuring source model validity, etc. Instead, developers only have to provide *strategies* containing *localized execution plans*, which are called by Pluto whenever they are needed during the execution of its transformation algorithm.

This general idea is applied in Figure 5, where instances of `Entity` are transformed to instances of `Relation`. The vertical arrow indicates the execution of the algorithm and the horizontal arrow shows the transformation of the model element. The transformation algorithm is explained in three steps: (1) *transformation preconditions*, (2) *transformation rules*, and (3) the *transformation protocol*.

Transformation Preconditions. For the transformation algorithm to run correctly, we need to make some assumptions about the source model:

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.

Transformation Rules. Given that our two validity rules, **V1** and **V2**, are satisfied, we can execute the desired transformation. The generic transformation algorithm of Pluto uses three rules to specify a *transformation order* among a set of transformable model elements:

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 below.

T2 Elements can only transform themselves *after* their parent and dependees have been transformed. Thus, an element may have to wait for other elements before it can transform itself,

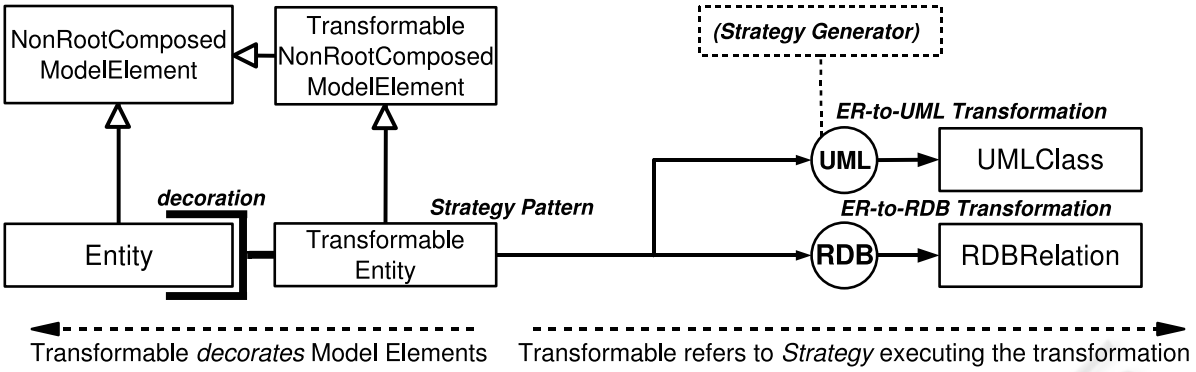

Figure 4: Reusable Transformation Concepts.

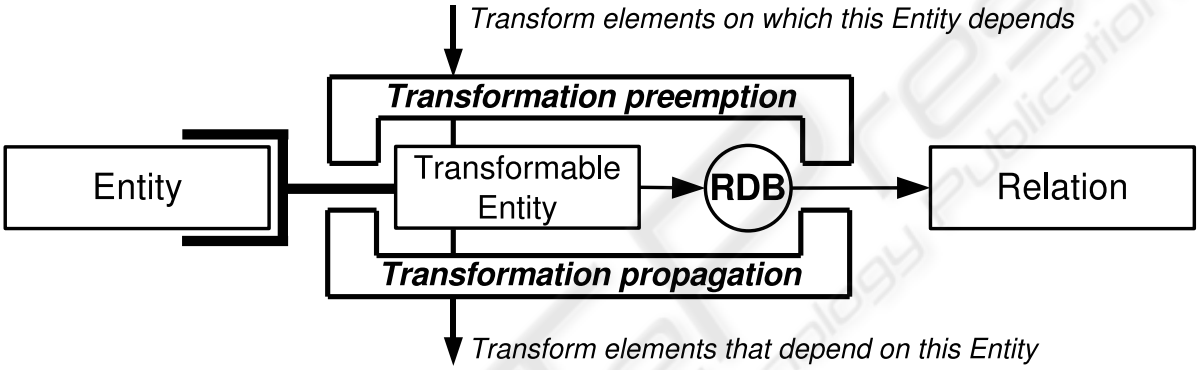

Figure 5: Strategies interact with the Pluto transformation algorithm.

which is called *transformation preemption*, as shown in Figure 5.

T3 After its transformation, a model element indirectly triggers the transformation of its children and dependants. This is called *transformation propagation* (see Fig. 5).

Two-Phase Transformation Protocol. Given our transformation rules, (**T1**–**T3**), by which concrete transformation strategies must abide, every model element can decide whether or not to transform itself based on *local* information, i.e. by determining whether its dependees and parent have been transformed (**T2**). Therefore, the transformation of a model element can be decomposed into two different transformation steps:

CT Conditional Transformation Request. 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**. Also, a **CT** request will propagate new **CT** requests to all the dependants and children of the model element after it has transformed itself (as required by **T3**). The implementation of the **CT** operation is *independent* of concrete transformations, so we have encapsulated this logic in the Pluto framework.

AT Actual Transformation. Unlike the **CT** phase, this second operation does not check preemption constraints and it does not propagate calls to children or dependants. Instead, it activates the execution plan of the strategy that was attached to the transformable model element, causing the target model to be manipulated. Therefore, this operation is private to the model element and it is called by **CT** when all preconditions have been satisfied, i.e. when the parent and the dependees of this model element have been transformed such that preemption is no longer required. This is the only phase that is specific to a model transformation, so these *actual transformations* must be provided by the developers by means of a concrete *transformation strategy* object, as explained in Section 6.1. The other steps of the algo-

rithm are transformation-independent, so developers can reuse them in concrete transformations without further configuration.

6.3 Illustration

Given a set of preconditions, a set of transformation rules, and the dichotomy between *conditional* and *actual* transformations, we now explain the generic execution strategy of our transformation algorithm by a simple example. Assume a conditional transformation request, **CT**, arrives at `ModelElement` in Figure 6 and assume that the model is *valid* according to the *transformation preconditions* **V1–V2**. This Section describes the steps taken by the transformation algorithm of Pluto in order to create a target model for `ModelElement` and its related elements. As noted above, this target model can be written in any other modeling language, depending on the contents of the concrete *transformation strategy* instances that were provided by the implementor of the transformation. This illustration is therefore independent of the chosen target language.

[Phases 1–6] Preemption of ModelElement. Pluto first checks whether `ModelElement` has any parent or dependees that have not executed their transformation, as required by **T2**. After discovering that $\texttt{Parent}_{ME}$ is not transformed, the transformation of `ModelElement` is preempted and a conditional transformation request **CT** is forwarded to $\texttt{Parent}_{ME}$. Pluto repeats the **CT** procedure for that parent and finds that $\texttt{Parent}_{ME}$ can transform immediately without being preempted (according to rule **T2**). Thus, the strategy attached to that parent is executed and a target element is created. Next, according to **T3**, the conditional transformation request must be propagated to all children, causing the call to reach `ModelElement`. Due to the existence of a dependee that still needs to be transformed, however, `ModelElement` is again preempted and a **CT** request is sent to $\texttt{Dependee}_{ME}$. The latter executes its transformation **AT** and propagates a **CT** request to its children.

[Phase 7] Transformation of ModelElement. Downward propagation of the **CT** request from $\texttt{Dependee}_{ME}$ eventually reaches `ModelElement`, one of the dependants of $\texttt{Dependee}_{ME}$, so Pluto checks whether `ModelElement` can transform itself. All preconditions have been satisfied, since both its parent and its dependee have successfully transformed, so `ModelElement` can execute its **AT** transformation according to **T2**. To do so, Pluto delegates to the *strategy* that was attached to the *decorator* of `ModelElement`. This strategy now manipulates the target model, possibly making use of the elements that were already created by its parent and dependee.

[Phases 8–9] Propagation from ModelElement. After the execution of `ModelElement` has completed, Pluto propagates a **CT** request to all the children and dependants, causing the transformation algorithm to visit $\texttt{Dependant}_{ME}$ and $\texttt{Child}_{ME}$. These elements can transform themselves without being preempted.

Due to space limitations, we cannot give a more realistic example of this transformation algorithm. Therefore, we refer to our technical report (Labey et al., 2007) for more information about the technical details of this algorithm in the context of multiple dependees or transitive dependencies between model elements.

7 RELATED WORK

In (Sunye et al., 2002) and (Varro and Pataricza, 2003), the applicability of Action Semantics (AS) (OMG, 2002) 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. One shortcoming is that cross-model transformations are not supported because AS is irreversibly linked to the Unified Modeling Language.

YATL (Patrascoiu, 2004) is a language for defining model transformations. It combines declarative concepts for querying the source model with imperative constructs for executing the transformation itself. By relying on the Meta Object Facility (The Meta Object Facility Core Specification 2.0, 2006), YATL provides support for pluggable metamodels, but the transformation tool does not offer reusable concepts for *defining* those metamodels. Furthermore, it is not clear how the transformation language can be applied to concrete instances of such newly introduced metamodels.

MTRANS (Peltier et al., 2001) 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. 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. Pluto, on the other hand, is independent of

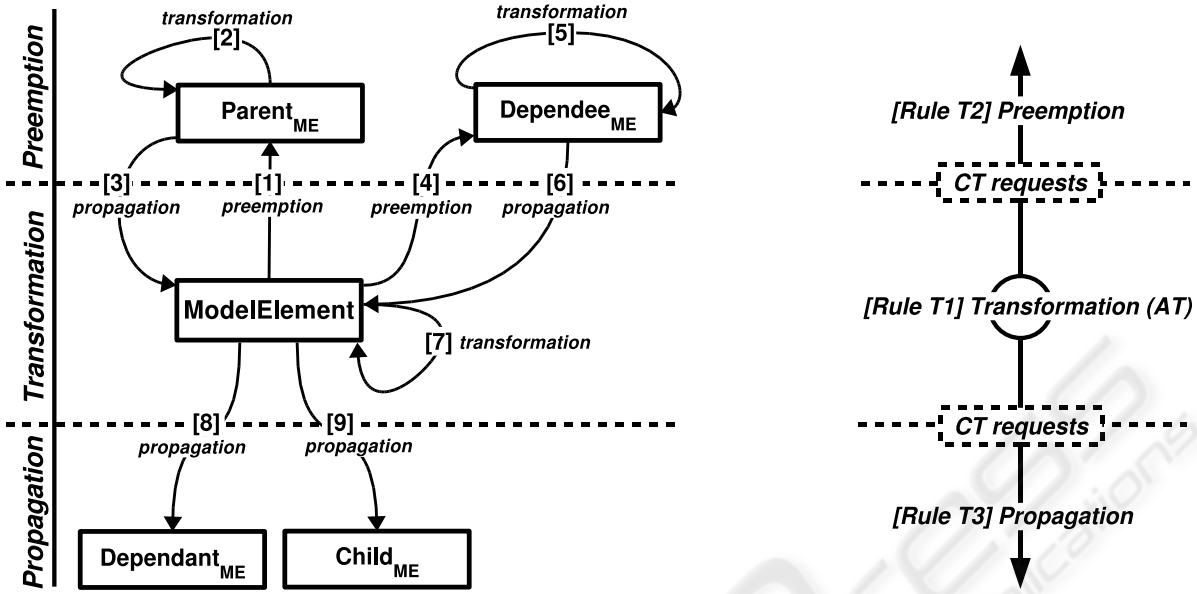

Figure 6: Illustration of Pluto's transformation algorithm.

any transformation language and transparently manages one-to-many and many-to-one dependencies for model transformations.

UMLX (Willink, 2003) and VMT (Sendall et al., 2003) 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 fits into a larger framework, Chameleon, (van Dooren, 2007), which can be used to *transform between programming languages*. Concrete examples of such transformations can be found in the work of (van Dooren and Steegmans, 2005; Delanote and Steegmans, 2006; Labey et al., 2007). By integrating Pluto'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.

Finally, a large number of transformation languages have been proposed, for example, Converge (Tratt and Clark, 2003) and the work of Kuznetsov (Kuznetsov, 2005). 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.

8 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. We have implemented Pluto, a framework providing 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.

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 while inheriting common modeling functionality from Pluto.

REFERENCES

BPEL4WS Language Specification: http://www.ibm.com/developerworks. (2003)

Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. (2003)

Rensink, A.: Subjects, Models, Languages, Transformations. In: Dagstuhl Seminar. (2005)

OMG UML Specification 1.5 http://www.omg.org/technology/documents/. (2003)

Mens, T., Czarnecki, K., Van Gorp, P.: A Taxonomy of Model Transformations. In: Dagstuhl Seminar. (2005)

The Meta Object Facility Core Specification 2.0: http://www.omg.org/technology/. (2006)

Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Reading, Massachusetts (1994)

Sven De Labey and Geert Delanote and Koen Vanderkimpen and Eric Steegmans: A Framework for Executing Cross-Model Transformations based on Pluggable Metamodels. In: Report CW489 K.U.Leuven – http://www.cs.kuleuven.be/publicaties/rapporten/. (2007)

Sunye, G., Le Guennec, A., et al.: Using UML Action Semantics for Model Execution and Transformation. In: The 13th IC on Advanced Information Systems Engineering. (2002)

Varro, D. and Pataricza, A.: UML Action Semantics for Model Transformation Systems. In: Periodica Politechnica. (2003)

Object Management Group: ptc/02-09-02: UML 1.5 – Action Semantics. (2002)

Patrascoiu, O.: YATL:Yet Another Transformation Language. In: Proceedings of the 1st European MDA Workshop, MDA-IA, University of Twente, the Nederlands (2004)

Peltier, M., Bezivin, J., Guillaume, G.: MTRANS: A General framework based on XSLT for model transformations. In: WTUMLO1. Proceedings of the First Workshop on Transformations in the Unified Modeling Language. (2001)

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)

Sendall, S., Perrouin, G., Guelfi, N., Biberstein, O.: Supporting Model-To-Model Transformations: the VMT Approach. In: Technical Report TR-CTIT-03-27, Twente. (2003)

van Dooren, M.: Abstractions for Improving, Creating, and Reusing Object-Oriented Programming Languages – PhD Thesis. K.U.Leuven. (2007)

van Dooren, M., Steegmans, E.: 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)

Delanote, G., Steegmans, E.: Concepts for abstracting away object reification at the level of platform independent models (PIMs). In: Proceedings of The Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software. (2006)

De Labey, S., van Dooren, M., Steegmans, E.: 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 (2007)

Tratt, L., Clark, T.: Model transformations in Converge. In: Workshop in Software Model Engineering (WiSME). (2003)

Kuznetsov, M.: Automated Model Transformation in MDA. In: Colloquium on Database and Information Systems. (2005)