# Model Transformations

Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio

Dipartimento di Informatica
Università degli Studi dell'Aquila
I-67100 L'Aquila, Italy
name.surname@univaq.it

**Abstract.** In recent years, Model-Driven Engineering has taken a leading role in advancing a new paradigm shift in software development. Leveraging models to a first-class status is at the core of this methodology. Shifting the focus of software development from coding to modeling permits programs to transform models in order to generate other models which are amenable for a wide range of purposes, including code generation. This paper introduces a classification of model transformation approaches and languages, illustrating the characteristics of the most prominent ones. Moreover, two specific application scenarios are proposed to highlight *bidirectionality* and *higher-order transformations* in the change propagation and coupled evolution domains, respectively.

## 1 Introduction

In recent years, Model-Driven Engineering [1] (MDE) has taken a leading role in advancing a new paradigm shift in software development. Leveraging models to a *first-class* status is at the core of this methodology. In particular, MDE proposes to extend the formal use of modelling languages in several interesting ways by adhering to the "everything is a model" principle [2]. Domains are analysed and engineered by means of metamodels, i.e., coherent sets of interrelated concepts. A model is said to conform to a metamodel, or in other words it is expressed in terms of the concepts formalized in the metamodel, constraints are expressed at the metalevel, and model transformations occur to produce target models out of source ones. Summarizing, these constitute a body of inter-related entities pursuing a common scope as in an ecosystem [3]. In this respect, model transformations represent the major gluing mechanism of the ecosystem by bridging different abstraction layers and/or views of a system. To this end, they require *"specialized support in several aspects in order to realize the full potential, for both the end-user and transformation developer"* [4].

In 2002 OMG issued the Query/View/Transformation Request For Proposal [5] in an attempt to define a standard transformation language. Although a final specification has been adopted at the end of 2005, the area of model transformation can still be considered in its infancy and further research is necessary

*a)* to investigate intrinsic characteristics of model transformation languages, such as bidirectionality, change propagation, and genericity;
*b)* to examine and devise transformation semantics, strategies and tools for testing and automatically verifying transformations; finally

M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.): SFM 2012, LNCS 7320, pp. 91–136, 2012.

*c)* to extend the scope of model transformation by assessing its full potential for new applications.

Interestingly, while *a)* and *b)* are analogous to what has been done in traditional programming research, *c)* is dealing with problems and needs which emerged over the last years during the adoption and deployment of MDE in industry. Since the beginning, model transformations have always been conceived as the essential mean to mainly transform models in order to generate artifacts considered very close to the final system (e.g., see [6–9] for the Web domain). However, lately specialized languages and techniques have been introduced to address more complex problems such as the coupled evolution which typically emerges during an MDE ecosystem life-cycle (e.g., [10–12]), or to manage simulation and fault detection in software systems (e.g., [13]).

In this paper, we summarize a classification of model transformation approaches and illustrate the main characteristics of prominent languages falling in this classification. Then, change propagation and coupled evolution are considered to illustrate complex application scenarios assessing the potential and significance of model transformations as described in the following.

**Change Propagation.** Change propagation and bidirectionality are relevant aspects in model transformations: often it is assumed that during development only the source model of a transformation undergoes modifications, however in practice it is necessary for developers to modify both the source and the target models of a transformation and propagate changes in both directions [14, 15]. There are two main approaches for realizing bidirectional transformations and supporting change propagation: by programming forward and backward transformations in any convenient unidirectional language and manually ensuring they are consistent; or by using a bidirectional transformation language where every program describes both a forward and a backward transformation simultaneously. A major advantage of the latter approach is that the consistency of the transformations can be guaranteed by construction.

**Metamodel/Model Coupled Evolution.** Evolution is an inevitable aspect which affects the whole life-cycle of software systems [16]. In general, artefacts can be subject to many kinds of changes, which range from requirements through architecture and design, to source code, documentation and test suites. Similarly to other software artefacts, metamodels can evolve over time too [17]. Accordingly, models need to be *co-adapted*[1] in order to remain compliant to the metamodel and not become eventually invalid. When manually operated the adaptation is error-prone and can give place to inconsistencies between the metamodel and the related artefacts. Such issue becomes very relevant when dealing with enterprise applications, since in general system models encompass a large population of instances which need to be appropriately adapted, hence inconsistencies can possibly lead to irremediable information erosion [18].

**Outline.** The structure of the paper is as follows. In Section 2 we review the basic concepts of Model-Driven Engineering, i.e., models, metamodels, and transformations.

[1] The terms (co-)adaptation, (co-)evolution, and coupled evolution will be used as synonyms throughout the paper, although in some approached the term coupled evolution denoted the parallel and coordinated evolution of two classes of artifacts.

Next section illustrates a number of approaches to model transformations and their characteristics, also prominent languages are outlined. Section 4 presents the Janus Transformation Language (JTL), a declarative model transformation language specifically tailored to support bidirectionality and change propagation. Section 5 proposes an approach based on higher-order model transformations (HOTs) to model coupled evolution. In particular, HOTs take a difference model formalizing the metamodel modifications and generate a model transformation able to adapt and recovery the validity of the compromised models. Section 6 draws some conclusions.

## 2 Model Driven Engineering

Model-Driven Engineering (MDE) refers to the systematic use of models as first-class entities throughout the software engineering life cycle. Model-driven approaches shift development focus from traditional programming language codes to models expressed in proper domain specific modeling languages. The objective is to increase productivity and reduce time to market by enabling the development of complex systems by means of models defined with concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain. This makes the models easier to specify, understand, and maintain [19] helping the understanding of complex problems and their potential solutions through abstractions.

The concept of Model Driven Engineering emerged as a generalization of the Model Driven Architecture (MDA) proposed by OMG in 2001 [20]. Kent [21] defines MDE on the base of MDA by adding the notion of software development process and modeling space for organizing models. Favre [22] proposes a vision of MDE where MDA is just one possible instance of MDE implemented by means of a set of technologies defined by OMG (MOF [23], UML [24], XMI [25], etc.) which provided a conceptual framework and a set of standards to express models, metamodels, and model transformations.

Even though MDA and MDE rely on *models* that are considered "first class citizens", there is no common agreement about what is a model. In [26] a model is defined as "a set of a statements about a system under study". Bézivin and Gerbé in [27] define a model as "a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system". According to Mellor et al. [28] a model "is a coherent set of formal elements describing something (e.g. a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis" such as communication of ideas between people and machines, test case generation, transformation into an implementation etc. The MDA guide [20] defines a model of a system as "a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language".

In MDE models are not considered as merely documentation but precise artifacts that can be understood by computers and can be automatically manipulated. In this scenario *metamodeling* plays a key role. It is intended as a common technique for defining the abstract syntax of models and the interrelationships between model elements. metamodeling can be seen as the construction of a collection of "concepts" (things, terms, etc.) within a certain domain. A model is an abstraction of phenomena in the

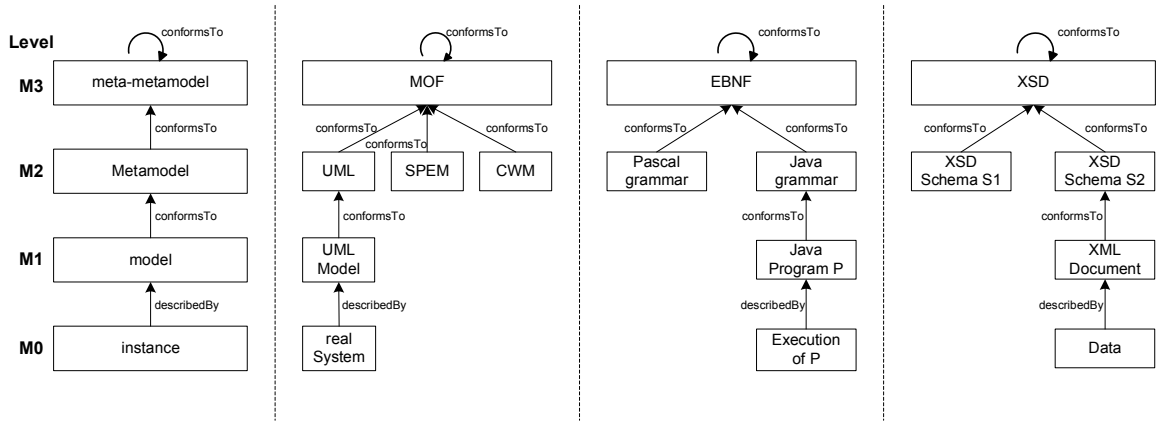



**Fig. 1.** The four layer metamodeling architecture

real world, and a metamodel is yet another abstraction, highlighting properties of the model itself. A model is said to *conform to* its *metamodel* like a program conforms to the grammar of the programming language in which it is written [2]. In this respect, OMG has introduced the four-level architecture shown in Fig. 1. At the bottom level, the `M0` layer is the real system. A model represents this system at level `M1`. This model conforms to its metamodel defined at level `M2` and the metamodel itself conforms to the metametamodel at level `M3`. The metametamodel conforms to itself. OMG has proposed MOF [23] as a standard for specifying metamodels. For example, the UML metamodel is defined in terms of MOF. A supporting standard of MOF is XMI [25], which defines an XML-based exchange format for models on the `M3`, `M2`, or `M1` layer. In EMF [29], Ecore is the provided language for specifying metamodels. This metamodeling architecture is common to other technological spaces as discussed by Kurtev et al. in [30]. For example, the organization of programming languages and the relationships between XML documents and XML schemas follows the same principles described above (see Fig. 1).

In addition to metamodeling, *model transformation* is also a central operation in MDE. While technologies such as MOF [23] and EMF [29] are well-established foundations on which to build metamodels, there is as yet no well-established foundation on which to rely in describing how we take a model and transform it to produce a target one. In the next section more insights about model transformations are given and after a brief discussion about the general approaches, the attention focuses on some of the today's available languages.

## 3 Model Transformations

The MDA guide [20] defines a model transformation as "the process of converting one model to another model of the same system". Kleppe et al. [31] defines a *transformation* as the automatic generation of a target model from a source model, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed to a model in the target language. A *transformation rule* is a description of how one or more

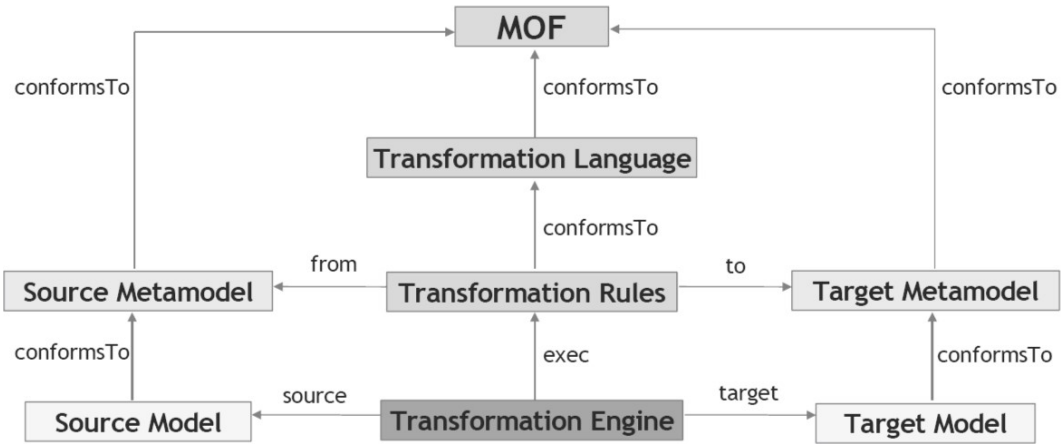



**Fig. 2.** Basic Concepts of Model Transformation

constructs in the source language can be transformed to one or more constructs in the target language.

Rephrasing these definitions by considering Fig. 2, a model transformation program takes as input a model conforming to a given source metamodel and produces as output another model conforming to a target metamodel. The transformation program, composed of a set of rules, should itself considered as a model. As a consequence, it is based on a corresponding metamodel, that is an abstract definition of the used transformation language.

Many languages and tools have been proposed to specify and execute transformation programs. In 2002 OMG issued the Query/View/Transformation request for proposal [5] to define a standard transformation language. Even though a final specification has been adopted at the end of 2005, the area of model transformation continues to be a subject of intense research. Over the last years, in parallel to the OMG process a number of model transformation approaches have been proposed both from academia and industry. The paradigms, constructs, modeling approaches, tool support distinguish the proposals each of them with a certain suitability for a certain set of problems.

In the following, a classification of the today's model transformation approaches is briefly reported, then some of the available model transformation languages are separately described. The classification is mainly based upon [32] and [33].

### 3.1 Classification

At top level, model transformation approaches can be distinguished between *model-to-model* and *model-to-text*. The distinction is that, while a model-to-model transformation creates its target as a model which conforms to the target metamodel, the target of a model-to-text transformation essentially consists of strings. In the following some classifications of model-to-model transformation languages discussed in [32] are described.

**Direct Manipulation Approach.** It offers an internal model representation and some APIs to manipulate it. It is usually implemented as an object oriented framework, which may also provide some minimal infrastructure. Users have to implement transformation

rules, scheduling, tracing and other facilities, mostly from the beginning in a programming language.

**Operational Approach.** It is similar to direct manipulation but offers more dedicated support for model transformations. A typical solution in this category is to extend the utilized metamodeling formalism with facilities for expressing computations. An example would be to extend a query language such as OCL with imperative constructs. Examples of systems in this category are QVT Operational mappings [34], XMF [35], MTL [36] and Kermeta [37].

**Relational Approach.** It groups declarative approaches in which the main concept is mathematical relations. In general, relational approaches can be seen as a form of constraint solving. The basic idea is to specify the relations among source and target element types using constraints that in general are non-executable. However, declarative constraints can be given executable semantics, such as in logic programming where predicates can be used to describe the relations. All of the relational approaches are side-effect free and, in contrast to the imperative direct manipulation approaches, create target elements implicitly. Relational approaches can naturally support multidirectional rules. They sometimes also provide backtracking. Most relational approaches require strict separation between source and target models, that is, they do not allow in-place update. Example of relational approaches are QVT Relations [34] and those enabling the specification of weaving models (like AMW [38]), which aim at defining rigorous and explicit correspondences between the artifacts produced during a system development [39]. Moreover, in [40] the application of logic programming has been explored for the purpose. Finally, in [41] we have investigated the application of the Answer Set Programming [42] for specifying relational and bidirectional transformations.

**Hybrid Approach.** It combines different techniques from the previous categories, like ATL [43] and ETL [44] that wrap imperative bodies inside declarative statements.

**Graph-Transformation Based Approach.** It draws on the theoretical work on graph tranformations. Describing a model transformation by graph transformation, the source and target models have to be given as graphs. Performing model transformation by graph transformation means to take the abstract syntax graph of a model, and to transform it according to certain transformation rules. The result is the syntax graph of the target model. Being more precise, graph transformation rules have an *LHS* and an *RHS* graph pattern. The *LHS* pattern is matched in the model being transformed and replaced by the *RHS* pattern in place. In particular, *LHR* represents the pre-condition of the given rule, while *RHS* describes the post-conditions. $LHR \cap RHS$ defines a part which has to exist to apply the rule, but which is not changed. $LHS - LHS \cap RHS$ defines the part which shall be deleted, and $RHS - LHS \cap RHS$ defines the part to be created. AGG [45] and AToM3 [46] are systems directly implementing the theoretical approach to attributed graphs and transformations on such graphs. They have built-in fixpoint scheduling with non-deterministic rule selection and concurrent application to all matching locations, and the rely on implicit scheduling by the user. The transformation rules are unidirectional and in-place. Systems such as VIATRA2 [47] and GReAT [48] extend the basic functionality of AGG and AToM3 by adding explicit scheduling. VIATRA2 users can

build state machines to schedule transformation rules whereas GReAT relies on data-flow graphs. Another interesting mean for transforming models is given by triple graph grammars (TGGs), which have been introduced by Schürr[49]. TGGs are a technique for defining the correspondence between two different types of models in a declarative way. The power of TGGs comes from the fact that the relation between the two models cannot only be defined, but the definition can be made operational so that one model can be transformed into the other in either direction; even more, TGGs can be used to synchronize and to maintain the correspondence of the two models, even if both of them are changed independently of each other; i.e., TGGs work incrementally. The main tool support for TGGs is Fujaba[2], which provided the foundation for MOFLON[3].

**Rule Based Approach.** Rule based approaches allow one to define multiple independent rules of the form *guard* => *action*. During the execution, rules are activated according to their guard not, as in more traditional languages, based on direct invocation [4]. When more than one rule is fired, more or less explicit management of such conflicting situation is provided, for instance in certain language a runtime error is raised. Besides the advantage of having an implicit matching algorithm, such approaches permit to encapsulate fragments of transformation logic within the rules which are self-contained units with crispy boundaries. This form of encapsulation is preparatory to any form of transformation composition [50].

### 3.2 Languages

In this section some of the languages referred above are singularly described. The purpose of the description is to provide the reader with an overiew of some existing model transformation languages.

**QVT.** In 2002 OMG issued the QVT RFP [5] describing the requirements of a standard language for the specification of model queries, views, and transformations according to the following definitions:

- A *query* is an expression that is evaluated over a model. The result of a query is one or more instances of types defined in the source model, or defined by the query language. Object Constraint Language (OCL 2.0) [51] is the query language used in QVT;
- A *view* is a model which is completely derived from a base model. A view cannot be modified separately from the model from which it is derived and changes to the base model cause corresponding changes to the view. If changes are permitted to the view then they modify the source model directly. The metamodel of the view is typically not the same as the metamodel of the source. A query is a restricted kind of view. Finally, views are generated via transformations;
- A *transformation* generates a target model from a source one. If the source and target metamodels are identical the transformation is called *endogeneous*. If they are different the transformation is called *exogeneous*. A model transformation may

[2] `http://www.fujaba.de`
[3] `http://www.moflon.org`

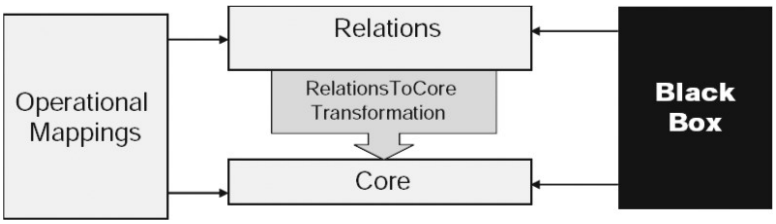



**Fig. 3.** QVT Architecture

also have several source models and several target models. A view is a restricted kind of transformation in which the target model cannot be modified independently from the source model. If a view is editable, the corresponding transformation must be bidirectional in order to reflect the changes back to the source model.

A number of research groups have been involved in the definition of QVT whose final specification has been reached at the end of November 2005 [34]. The abstract syntax of QVT is defined in terms of MOF 2.0 metamodel. This metamodel defines three sub-languages for transforming models. OCL 2.0 is used for querying models. Creation of views on models is not addressed in the proposal.

The QVT specification has a hybrid declarative/imperative nature, with the declarative that forms the framework for the execution semantics of the imperative part. By referring to Fig. 3, the layers of the declarative part are the following:

- A user-friendly *Relations* metamodel which supports the definition of complex object pattern matching and object template creation;
- A *Core* metamodel defined using minimal extensions to EMOF and OCL.

By referring to [34], a relation is a declarative specification of the relationships between MOF models. The *Relations* language supports complex object pattern matching, and implicitly creates trace classes and their instances to record what occurred during a transformation execution. Relations can assert that other relations also hold between particular model elements matched by their patterns. Finally, *Relations* language has a graphical syntax.

Concerning the *Core* it is a small model/language which only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. It treats all of the model elements of source, target and trace models symmetrically. It is equally powerful to the *Relations* language, and because of its relative simplicity, its semantics can be defined more simply, although transformation descriptions described using the *Core* are therefore more verbose. In addition, the trace models must be explicitly defined, and are not deduced from the transformation description, as is the case with *Relations*. The core model may be implemented directly, or simply used as a reference for the semantics of *Relations*, which are mapped to the Core, using the transformation language itself.

To better clarify the conceptual link between *Relations* and *Core* languages, an analogy can be drawn with the Java architecture, where the Core language is like Java Byte Code and the Core semantics is like the behavior specification for the Java Virtual Machine. The Relations language plays the role of the Java language, and the standard

transformation from Relations to Core is like the specification of a Java Compiler which produces Byte Code.

Sometimes it is difficult to provide a complete declarative solution to a given transformation problem. To address this issue QVT proposes two mechanisms for extending the declarative languages *Relations* and *Core*: a third language called *Operational Mappings* and a mechanism for invoking transformation functionality implemented in an arbitrary language (*Black Box*).

The *Operational Mappings* language is specified as a standard way of providing imperative implementations. It provides OCL extensions with side effects that allow a more procedural style, and a concrete syntax that looks familiar to imperative programmers. A transformation entirely written using Operation Mappings is called an "operational transformation".

The *Black Box* mechanism makes possible to "plug-in" and execute external code. This permits to implement complex algorithms in any programming language, and reuse already available libraries.

**AGG.** AGG [45] is a development environment for attributed graph transformation systems supporting an algebraic approach to graph transformation. It aims at specifying and rapid prototyping applications with complex, graph structured data. AGG supports typed graph transformations including type inheritance and multiplicities. It may be used (implicitly in "code") as a general purpose graph transformation engine in high-level JAVA applications employing graph transformation methods. The source, target, and common metamodels are represented by typed graphs. Graphs may additionally be attributed using Java code. Model transformations are specified by graph rewriting rules that are applied non-deterministically until none of them can be applied anymore. If an explicit application order is required, rules can be grouped in ordered layers. AGG features rules with negative application conditions to specify patterns that prevent rule executions. Finally, AGG offers validation support that is consistency checking of graphs and graph transformation systems according to graph constraints, critical pair analysis to find conflicts between rules (that could lead to a non-deterministic result) and checking of termination criteria for graph transformation systems. An available tool support provides graphical editors for graphs and rules and an integrated textual editor for Java expressions. Visual interpretation and validation of transformations are also supported.

**ATL.** ATL (ATLAS Transformation Language) [43] is a hybrid model transformation language containing a mixture of declarative and imperative constructs. The former allows to deal with simple model transformations, while the imperative part helps in coping with transformation of higher complexity. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. During the execution of a transformation source models may be navigated but changes are not allowed. Target models cannot be navigated.

ATL transformations are specified in terms of *modules*. A module contains a mandatory *header* section, *import* section, and a number of *helpers* and *transformation rules*. Header section gives the name of a transformation module and declares the source and target models (e.g., see lines 1-2 in Fig. 4). The source and target models are typed by their metamodels. The keyword `create` indicates the target model, whereas the keyword `from` indicates the source model. In the example of Fig. 4 the target model bound

```
1 module PetriNet2PNML;
2 create OUT : PNML from IN : PetriNet;
3 ...
4 rule Place {
5         from
6                 e : PetriNet!Place
7                 --(guard)
8         to
9                 n : PNML!Place
10                (
11                        name <- e.name,
12                        id <- e.name,
13                        location <- e.location
14                ),
15                name : PNML!Name
16                (
17                        labels <- label
18                ),
19                label : PNML!Label
20                (
21                        text <- e.name
22                )
23 }
```

**Fig. 4.** Fragment of a declarative ATL transformation

to the variable `OUT` is created from the source model `IN`. The source and target metamodels, to which the source and target model conform, are `PetriNet` and `PNML` [52], respectively.

Helpers and transformation rules are the constructs used to specify the transformation functionality. Declarative ATL rules are called *matched rules*. They specify relations between *source patterns* and *target patterns*. The name of a rule is given after the keyword `rule`. The source pattern of a rule (lines `5-7`, Fig. 4) specifies a set of *source types* and an optional *guard* given as a Boolean expression in OCL. A source pattern is evaluated on a set of matches in the source models. The target pattern (lines `8-22`, Fig. 4) is composed of a set of *elements*. Each of these elements (e.g., the one at lines `9-14`, Fig. 4) specifies a *target type* from the target metamodel (e.g., the type `Place` from the `PNML` metamodel) and a set of *bindings*. A binding refers to a feature of the type (i.e. an attribute, a reference or an association end) and specifies an expression whose value is used to initialize that feature. In some cases complex transformation algorithms may be required and it may be difficult to specify them in a pure declarative way. For this issue ATL provides two imperative constructs: *called rules*, and *action blocks*. A called rule is a rule called by other ones like a procedure. An action block is a sequence of imperative instructions that can be used in either matched or called rules. The imperative statements in ATL are the well-known constructs for specifying control flow such as conditions, loops, assignments, etc.

**ETL.** Similarly to ATL, ETL [44] (Epsilon Transformation Language) is a hybrid model transformation language that has been developed atop the infrastructure provided by the Epsilon model management platform [53]. By building on Epsilon, ETL achieves syntactic and semantic consistency and enhanced interoperability with a number of additional languages, also been built atop Epsilon, and which target tasks such as model-to-text transformation, model comparison, validation, merging and unit testing.

ETL enables the specification of transformations that can transform an arbitrary number of source models into an arbitrary number of target models. ETL transformations are given in terms of modules. An ETL module can import a number of other ETL modules. In this case, the importing ETL module inherits all the rules and pre/post blocks specified in the modules it imports (recursively).

**GReAT.** GReAT [48] (Graph Rewriting and Transformation Language) is a graph-transformation language that supports the high-level specification of complex model transformation programs. In this language, one describes the transformations as sequenced graph rewriting rules that operate on the input models and construct an output model. The rules specify complex rewriting operations in the form of a matching pattern and a subgraph to be created as the result of the application of the rule. The rules *i)* always operate in a context that is a specific subgraph of the input, and *ii)* are explicitly sequenced for efficient execution. The rules are specified visually using a graphical model builder tool. GReAT can be divided into three distinct parts:

- *Pattern specification language.* This language is used to express complex patterns that are matched to select elements in the current graph. The pattern specification language uses a notion of cardinality on each pattern vertex and each edge;
- *Graph transformation language.* It is a rewriting language that uses the pattern language described above. It treats the source model, the target model and temporary objects as a single graph that conforms to a unified metamodel. Each pattern object's type conforms to this metamodel and only transformations that do not violate the metamodel are allowed. At the end of the transformation, the temporary objects are removed and the two models conform exactly to their respective metamodels. Guards to manage the rule applications can be specified as boolean C++ expressions;
- *Control flow language.* It is a high-level control flow language that can control the application of the productions and allow users to manage the complexity of the transformations. In particular, the language supports a number of features: *(i) Sequencing*, rules can be sequenced to fire one after another, *(ii) Non-Determinism*, rules can be specified to be executed "in parallel", where the order of firing of the parallel rules is non deterministic, *(iii) Hierarchy*, compound rules can contain other compound rules or primitive rules, *(iv) Recursion*, a high level rule can call itself, *(v) Test/Case*, a conditional branching construct that can be used to choose between different control flow paths.

**VIATRA2.** VIATRA2 [47] is an Eclipse-based general-purpose model transformation engineering framework intended to support the entire life-cycle for the specification, design, execution, validation and maintenance of transformations within and between various modelling languages and domains.

Its rule specification language is a unidirectional transformation language based mainly on graph transformation techniques that combines the graph transformation and Abstract State Machines [54] into a single paradigm. Being more precise, in VIATRA2 the basic concept to define model transformations is the (graph) pattern. A pattern is a collection of model elements arranged into a certain structure fulfilling additional constraints (as defined by attribute conditions or other patterns). Patterns can be matched

on certain model instances, and upon successful pattern matching, elementary model manipulation is specified by graph transformation rules. There is no predefined order of execution of the transformation rules. Graph transformation rules are assembled into complex model transformations by abstract state machine rules, which provide a set of commonly used imperative control structures with precise semantics. This permits to collocate VIATRA2 as a hybrid language since the transformation rule language is declarative but the rules cannot be executed without an execution strategy specified in an imperative manner.

Important specification features of VIATRA2 include recursive (graph) patterns, negative patterns with arbitrary depth of negation, and generic and meta-transformations (type parameters, rules manipulating other rules) for providing reuse of transformations [55].

## 4 Application Scenario 1: Change Propagation with JTL

Bidirectionality and change propagation are relevant aspects in model transformations: often it is assumed that during development only the source model of a transformation undergoes modifications, however in practice it is necessary for developers to modify both the source and the target models of a transformation and propagate changes in both directions [14, 15]. There are two main approaches for realizing bidirectional transformations: by programming forward and backward transformations in any convenient unidirectional language and manually ensuring they are consistent; or by using a bidirectional transformation language where every program describes both a forward and a backward transformation simultaneously. A major advantage of the latter approach is that the consistency of the transformations can be guaranteed by construction. Moreover, source and target roles are not fixed since the transformation direction entails them. Therefore, considerations made about the mapping executed in one direction are completely equivalent to the opposite one.

The relevance of bidirectionality in model transformations has been acknowledged already in 2005 by the Object Management Group (OMG) by including a bidirectional language in their Query View Transformation (QVT) [56]. Unfortunately, as pointed out by Perdita Stevens in [57] the language definition is affected by several weaknesses. Therefore, while MDE requirements demand enough expressiveness to write non-bijective transformations [58], the QVT standard does not clarify how to deal with corresponding issues, leaving their resolution to tool implementations. Moreover, a number of approaches and languages have been proposed due to the intrinsic complexity of bidirectionality. Each language is characterized by a set of specific properties pertaining to a particular applicative domain [32].

This section outlines the Janus Transformation Language (JTL), a declarative model transformation language specifically tailored to support bidirectionality and change propagation. In particular, the distinctive characteristics of JTL are

– *non-bijectivity*, non-bijective bidirectional transformations are capable of mapping a model into a set of models, as for instance when a single change in a target model might semantically correspond to a family of related changes in more than one source model. JTL provides support to non-bijectivity and its semantics assures

that all the models are computed at once independently whether they represent the outcome of the backward or forward execution of the bidirectional transformation;
- *model approximation*, generally transformations are not total which means that target models can be manually modified in such a way they are not reachable anymore by any forward transformation, then traceability information are employed to back propagate the changes from the modified targets by inferring the *closest* model that approximates the ideal source one at best.

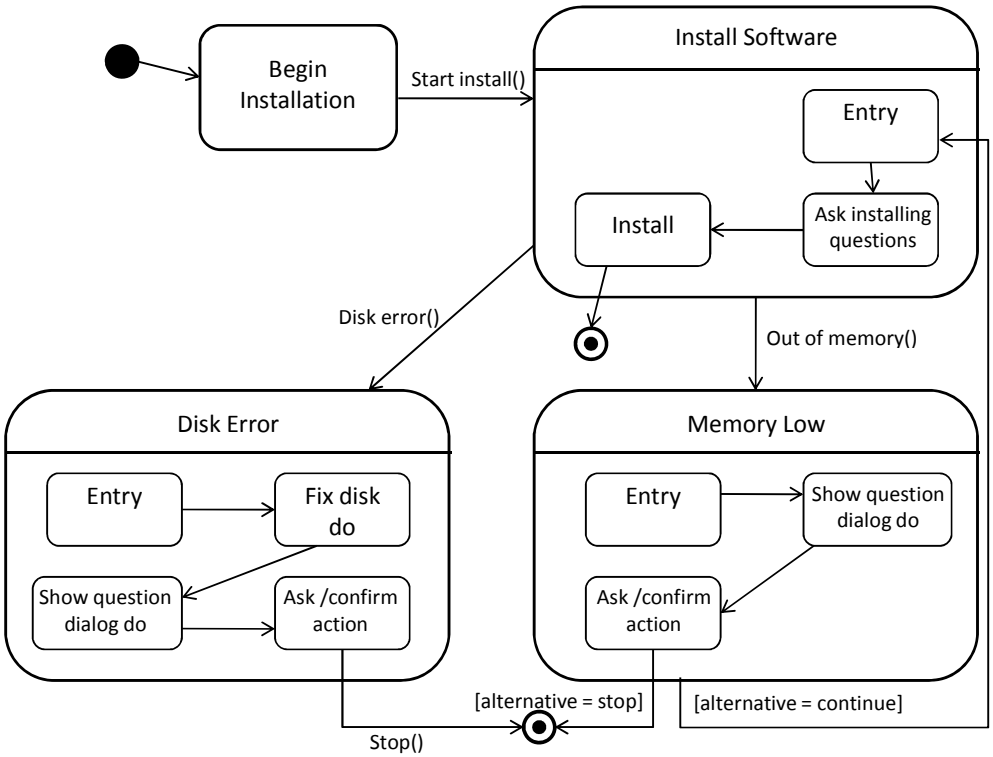
The language expressiveness and applicability have been validated by implementing a number of model transformations. In this section we focus on the *Collapse/Expand State Diagrams* benchmark which have been defined in the *GRACE International Meeting on Bidirectional Transformations* [59] to compare and assess different bidirectional approaches. The JTL semantics is defined in terms of the Answer Set Programming (ASP) [42], a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. Bidirectional transformations are translated via semantic anchoring [60] into search problems which are reduced to computing stable models, and the DLV solver [61] is used to perform search.

### 4.1 Motivating Scenario

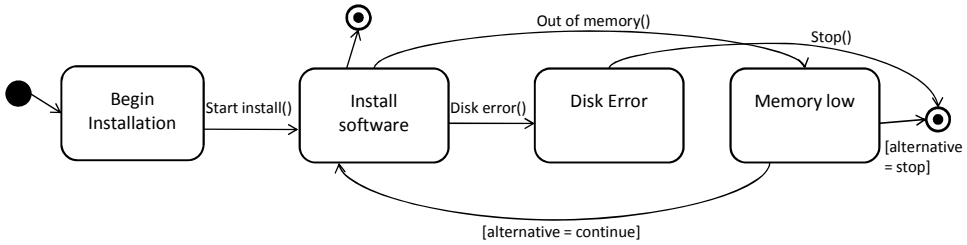
Let us consider the *Collapse/Expand State Diagrams* benchmark defined in [59]: starting from a hierarchical state diagram (involving some one-level nesting) as the one reported in Fig. 5.a, a flat view has to be provided as in Fig. 5.b. Furthermore, any manual modifications on the (target) flat view should be back propagated and eventually reflected in the (source) hierarchical view. For instance, let us suppose the designer modifies the flat view by changing the name of the initial state from `Begin Installation` to `Start Install shield` (see $\Delta_1$ change in Figure 6). Then, in order to persist such a refinement to new executions of the transformation, the hierarchical state machine has to be consistently updated by modifying its initial state as illustrated in Fig. 7.

The flattening is a non-injective operation requiring specific support to back propagate modifications operated on the flattened state machine to the nested one. For instance, the flattened view reported in Fig. 5 can be extended by adding the alternative `try again` from the state `Disk Error` to `Install software` (see $\Delta_2$ changes in Fig. 6). This gives place to an interesting situation: the new transition can be equally mapped to each one of the nested states within `Install Software` as well as to the container state itself. Consequently, more than one source model propagating the changes exists[4]. Intuitively, each time hierarchies are flattened there is a loss of information which causes ambiguities when trying to map back corresponding target revisions. Some of these problems can be alleviated by managing traceability information of the transformation executions which can be exploited later on to trace back the changes: like this each generated element can be linked with the corresponding source and contribute to the resolution of some of the ambiguities. Nonetheless, traceability is a necessary

[4] It is worth noting that the case study and examples have been kept deliberately simple since they suffice to show the relevant issues related to non-bijectivity.




a) A sample Hierarchical State Machine (HSM).




b) The corresponding Non-Hierarchical State Machine (NHSM).

**Fig. 5.** Sample models for the *Collapse/Expand State Diagrams* benchmark

but not sufficient condition to support bidirectionality, since for instance elements discarded by the mapping may not appear in the traces, as well as new elements added on the target side. For instance, the generated flattened view in Fig. 5.b can be additionally manipulated through the $\Delta_3$ revisions which consist of adding some extra-functional information for the `Install Software` state and the transition between from `Memory low` and `Install Software` states. Because of the limited expressive power of the hierarchical state machine metamodel which does not support extra-functional annotations, the $\Delta_3$ revisions do not have counterparts in the state machine in Fig. 7.

Current declarative bidirectional languages, such as QVT relations (QVT-R), are often ambivalent when discussing non-bijective transformations as already pointed out in [57]. Other approaches, notably hybrid or graph-based transformation techniques, even if claiming to support bidirectionality, are able to deal only with (partially) bijective mappings [14]. As a consequence, there is not a clear understanding of what

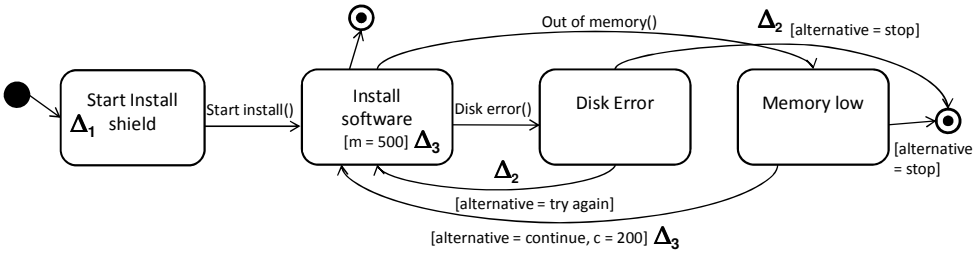



**Fig. 6.** A revision of the generated non-hierarchical state machine

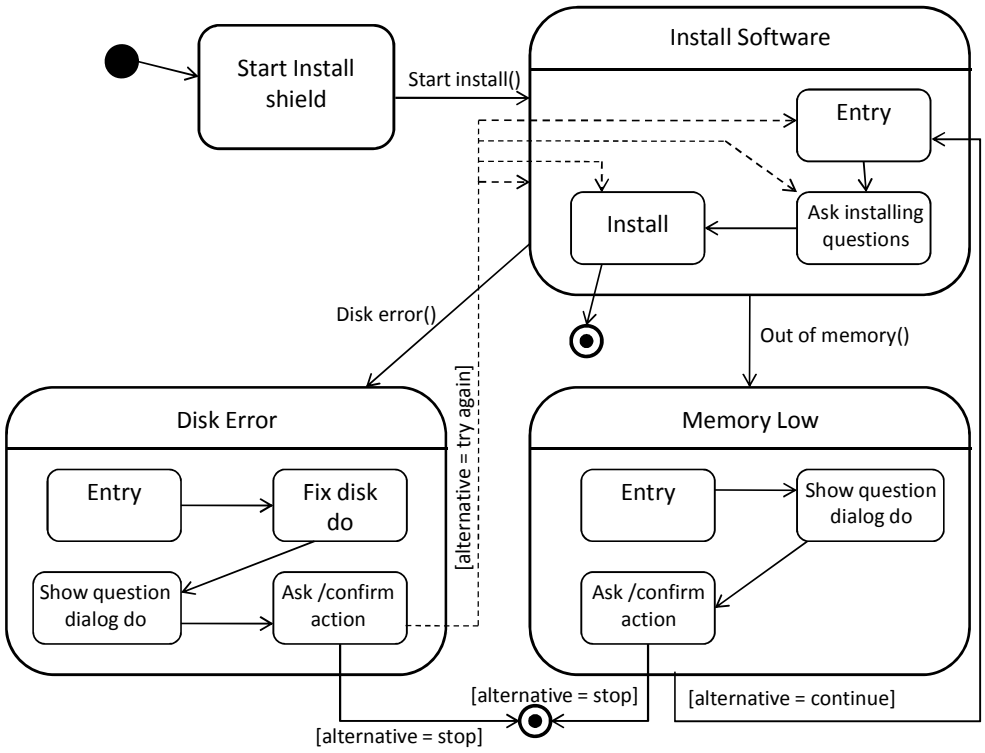



**Fig. 7.** The source hierarchical state machine synchronised with the target changes

non-bijectivity implies causing language implementors to adopt design decisions which differ from an implementation to another.

In order to better understand how the different languages deal with non-bijectivity, we have specified the hierarchical to non-hierarchical state machines transformation (HSM2NHSM) by means of the Medini[5] and MOFLON[6] systems. The former is an implementation of the QVT-R transformation language, whereas the latter is a framework which bases on Triple Graph Grammars (TGGs) [49]: our experience with them is outlined in the following

*Medini.* When trying to map the generated target model back to the source without any modification, a new source model is generated which differs from the original one[7].

[5] `http://projects.ikv.de/qvt/`

[6] `http://www.moflon.org`

[7] The interested reader can access the full implementation of both the attempts at the following address `http://www.mrtc.mdh.se/~acicchetti/HSM2NHSM.php`

In particular, incoming (outgoing) transitions to (from) nested states are flattened to the corresponding parent: when going back such mapping makes the involved nested states to disappear (as `Entry` and `Install` in the `Install Software` composite in Fig. 5). Moreover, the same mapping induces the creation of extra composite states for existing simple states, like `Begin Installation` and the initial and final states of the hierarchical state machine. Starting from this status, we made the modifications on the target model as prescribed by Fig. 6 and re-applied the transformation in the source direction, i.e. backward. In this case, the `Start Install shield` state is correctly mapped back by renaming the existing `Begin Installation` in the source. In the same way, the modified transition from `Disk Error` to the final state is consistently updated. However, the newly added transition outgoing from `Disk Error` to `Install software` is mapped by default to the composite state, which might not be the preferred option for the user. Finally, the manipulation of the attributes related to memory requirements and cost are not mapped back to any source element but are preserved when new executions of the transformation in the target direction are triggered.

*MOFLON.* The TGGs implementation offered by MOFLON is capable of generating Java programs starting from diagrammatic specifications of graph transformations. The generated code realizes two separate unidirectional transformations which as in other bidirectional languages should be consistent by construction. However, while the forward transformation implementation can be considered complete with respect to the transformation specification, the backward program restricts the change propagation to attribute updates and element deletions. In other words, the backward propagation is restricted to the contexts where the transformation can exploit trace information.

In the next sections, we firstly motivate a set of requirements a bidirectional transformation language should meet to fully achieve its potential; then, we introduce the JTL language, its support to non-bijective bidirectional transformations, and its ASP-based semantics.

### 4.2 Requirements for Bidirectionality and Change Propagation

This section refines the definition of bidirectional model transformations as proposed in [57] by explicitly considering non-bijective cases. Even if some of the existing bidirectional approaches enable the definition of non-bijective mappings [57, 15], their validity is guaranteed only on bijective sub-portions of the problem. As a consequence, the forward transformation can be supposed to be an injective function, and the backward transformation its corresponding inverse; unfortunately, such requirement excludes most of the cases [62]. In general, a bidirectional transformation *R* between two classes of models, say *M* and *N*, and *M* more expressive than *N*, is characterized by two unidirectional transformations

$$\overrightarrow{R} : M \times N \rightarrow N$$
$$\overleftarrow{R} : M \times N \rightarrow M^*$$

where $\overrightarrow{R}$ takes a pair of models *(m, n)* and works out how to modify *n* so as to enforce the relation $\overrightarrow{R}$. In a similar way, $\overleftarrow{R}$ propagates changes in the opposite direction: $\overleftarrow{R}$ is a non-bijective function able to map the target model in a set of corresponding

source models conforming to $M$[8]. Furthermore, since transformations are not total in general, bidirectionality has to be provided even in the case the generated model has been manually modified in such a way it is not reachable anymore by the considered transformation. Traceability information is employed to back propagate the changes from the modified targets by inferring the *closest*[9] model that approximates the ideal source one at best. More formally the backward transformation $\overleftarrow{R}$ is a function such that:

*(i)* if *R(m,n)* is a non-bijective consistency relation, $\overleftarrow{R}$ generates all the resulting models according to R;

*(ii)* if *R(m,n)* is a non-total consistency relation, $\overleftarrow{R}$ is able to generate a result model which approximates the ideal one.

This definition alone does not constrain much on the behavior of the reverse transformation and additional requirements are necessary in order to ensure that the propagation of changes behaves as expected.

*Reachability.* In case a generated model has been manually modified ($n'$), the backward transformation $\overleftarrow{R}$ generates models ($m^*$) which are exact, meaning that the original target may be reached by each of them via the transformation without additional side effects. Formally:

$$\overleftarrow{R}(m, n') = m^* \in M^*$$

$$\overrightarrow{R}(m', n') = n' \in N \text{ for each } m' \in m^*$$

*Choice preservation.* Let $n'$ be the target model generated from an arbitrary model $m'$ in $m^*$ as above: when the user selects $m'$ as the appropriate source pertaining to $n'$ the backward transformation has to generate exactly $m'$ from $n'$ disregarding the other possible alternatives $t \in m^*$ such that $t \neq m'$. In other words, a valid round-trip process has to be guaranteed even when multiple sources are available [63]:

$$\overleftarrow{R}(m', \overrightarrow{R}(m', n')) = m' \text{ for each } m' \in m^*$$

Clearly, the above requirement in order to be met demands for adequate traceability information management.

In the rest of the paper, the proposed language is introduced and shown to satisfy the above requirements. The details of the language and its supporting development environment are presented in Section 4.3, whereas in Section 4.4 the usage of the language is demonstrated by means of the benchmark case.

[8] For the sake of readability, we consider a non-bijective backward transformation assuming that only $M$ contains elements not represented in $N$. However, the reasoning is completely analogous for the forward transformation and can be done by exchanging the roles of $M$ and $N$.

[9] This concept is clarified in Sect. 4.3, where the transformation engine and its derivation mechanism are discussed.

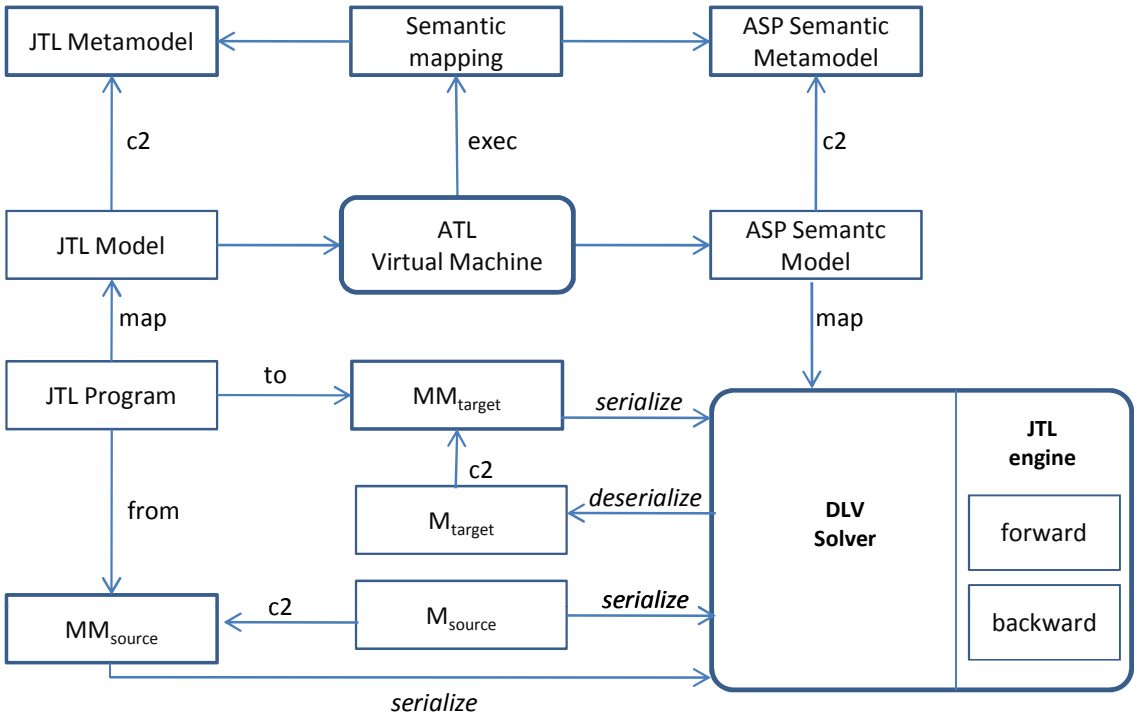



**Fig. 8.** Architecture overview of the JTL environment

### 4.3 The Janus Transformation Language

The Janus Transformation Language (JTL) is a declarative model transformation language specifically tailored to support bidirectionality and change propagation. The implementation of the language relies on the Answer Set Programming (ASP) [42]. This is a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. Being more precise model transformations specified in JTL are transformed into ASP programs (search problems), then an ASP solver is executed to find all the possible stable models that are sets of atoms which are consistent with the rules of the considered program and supported by a deductive process.

The overall architecture of the environment supporting the execution of JTL transformations is reported in Fig. 8. The *JTL engine* is written in the ASP language and makes use of the *DLV solver* [61] to execute transformations in both forward and backward directions. The engine executes JTL transformations which have been written in a QVT-like syntax, and then automatically transformed into ASP programs. Such a semantic anchoring has been implemented in terms of an ATL [43] transformation defined on the JTL and ASP metamodels. Also the source and target metamodels of the considered transformation ($MM_{source}$, $MM_{target}$) are automatically encoded in ASP and managed by the engine during the execution of the considered transformation and to generate the output models.

The overall architecture has been implemented as a set of plug-ins of the Eclipse framework and mainly exploits the Eclipse Modelling Framework (EMF) [29] and the

ATLAS Model Management Architecture (AMMA) [64]. Moreover, the DLV solver has been wrapped and integrated in the overall environment. In the rest of the section all the components of the architecture previously outlined are presented in detail.

**The Janus Transformation Engine.** As previously said the Janus transformation engine is based on a relational and declarative approach implemented using the ASP language to specify bidirectional transformations. The approach exploits the benefits of logic programming that enables the specification of relations between source and target types by means of predicates, and intrinsically supports bidirectionality [32] in terms of unification-based matching, searching, and backtracking facilities.

Starting from the encoding of the involved metamodels and the source model (see the *serialize* arrows in the Fig. 8), the representation of the target one is generated according to the JTL specification. The computational process is performed by the JTL engine (as depicted in Figure 8) which is based on an ASP bidirectional transformation program executed by means of an ASP solver called DLV [61].

**Encoding of Models and Metamodels.** In the proposed approach, models and metamodels are defined in a declarative manner by means of a set of logic assertions. In particular, they are considered as graphs composed of nodes, edges and properties that qualify them. The metamodel encoding is based on a set of *terms* each characterized by the predicate symbols `metanode`, `metaedge`, and `metaprop`, respectively. A fragment of the hierarchical state machine metamodel considered in Section 4.1 is encoded in Listing 1.1. For instance, the `metanode(HSM,state)` in line `1` encodes the metaclass `state` belonging to the metamodel `HSM`. The `metaprop(HSM,name,state)` in line `3` encodes the attribute named `name` of the metaclass `state` belonging to the metamodel `HSM`. Finally, the `metaedge(HSM,association,source,transition,state)` in line `6` encodes the association between the metaclasses `transition` and `state`, typed `association`, named `source` and belonging to the metamodel `HSM`. The terms induced by a certain metamodel are exploited for encoding models conforming to it. In particular, models are sets of entities (represented through the predicate symbol `node`), each characterized by properties (specified by means of `prop`) and related together by relations (represented by `edge`). For instance, the state machine model in Fig. 5 is encoded in the Listing 1.2. In particular, the `node(HSM,"s1",state)` in line `1` encodes the instance identified with `"s1"` of the class `state` belonging to the metamodel `HSM`. The `prop(HSM,"s1",name,"start")` in line `4` encodes the attribute

```
1 metanode(HSM, state).
2 metanode(HSM, transition).
3 metaprop(HSM, name, state).
4 metaprop(HSM, trigger, transition).
5 metaprop(HSM, effect, transition).
6 metaedge(HSM, association, source, transition, state).
7 metaedge(HSM, association, target, transition, state).
8 [...]
```

**Listing 1.1.** Fragment of the State Machine metamodel

```
1 node(HSM, "s1", state).
2 node(HSM, "s2", state).
3 node(HSM, "t1", transition).
4 prop(HSM,"s1.1","s1",name,"begin␣installation").
5 prop(HSM,"s2.1","s2",name,"install␣software").
6 prop(HSM,"t1.1","t1",trigger,"install␣software").
7 prop(HSM,"t1.2","t1",effect,"start␣install").
8 edge(HSM,"tr1",association,source, "s1","t1").
9 edge(HSM,"tr1",association,target, "s2","t1").
10 [...]
```

**Listing 1.2.** Fragment of the State Machine model in Figure 5

`name` of the class `"s1"` with value `"start"` belonging to the metamodel `HSM`. Finally, the `edge(HSM,"tr1",association,source,"s1","t1")` in line `7` encodes the instance `"tr1"` of the association between the state `"s1"` and the transition `"t1"` belonging to the metamodel `HSM`.

**Model Transformation Execution.** After the encoding phase, the deduction of the target model is performed according to the rules defined in the ASP program. The transformation engine is composed of *i) relations* which describe correspondences among element types of the source and target metamodels, *ii) constraints* which specify restrictions on the given relations that must be satisfied in order to execute the corresponding mappings, and an *iii) execution engine* (described in the rest of the section) consisting of bidirectional rules implementing the specified relations as executable mappings. Relations and constraints are obtained from the given JTL specification, whereas the execution engine is always the same and represents the bidirectional engine able to interpret the correspondences among elements and execute the transformation. The transformation process logically consists of the following steps:

(i) given the input (meta)models, the execution engine induces all the possible solution candidates according to the specified relations;
(ii) the set of candidates is refined by means of constraints.

Listing 1.3 contains a fragment of the ASP code implementing relations and constraints of the HSM2NHSM transformation discussed in Section 4.1. In particular, the terms in lines `1-2` define the relation called `"r1"` between the metaclass `State machine` belonging to the `HSM` metamodel and the metaclass `State machine` belonging to the `NHSM` metamodel. An ASP constraint expresses an invalid condition: for example, the constraints in line `3-4` impose that each time a state machine occurs in the source model it has to be generated also in the target model. In fact, if each atoms in its body is true then the correspondent solution candidate is eliminated. Similarly, the relation between the metaclasses `State` of the involved metamodels is encoded in line `6-7`. In this case, constraints in line `8-11` impose that each time a state occurs in the HSM model, the correspondent one in the NHSM model is generated only if the source element is not a sub-state, vice versa, each state in the NHSM model is mapped into the HSM model. Finally, the relation between the metaclasses `Composite state` and `State` is encoded

```
1 relation ("r1",  HSM, stateMachine).
2 relation ("r1", NHSM, stateMachine).
3 :- node(HSM, "sm1", stateMachine), not  node'(HSM, "sm1", stateMachine).
4 :- node(NHSM, "sm1", stateMachine), not  node'(NHSM, "sm1", stateMachine).
5
6 relation ("r2", HSM, state).
7 relation ("r2", NHSM, state).
8 :- node(HSM, "s1", state), not edge(HSM, "ow1", owningCompositeState, "s1", "cs1
      "), not node'(NHSM, "s1", state).
9 :- node(HSM, "s1", state), edge(HSM, "ow1", owningCompositeState, "s1", "cs1"),
     node(HSM, "cs1", compositeState), node'(NHSM, "s1", state).
10 :- node(NHSM, "s1", state), not trace_node(HSM, "s1", compositeState), not node
      '(HSM, "s1", state).
11 :- node(NHSM, "s1", state), trace_node(HSM, "s1", compositeState), node'(HSM, "
      s1", state).
12
13 relation ("r3", HSM, compositeState).
14 relation ("r3", NHSM, state).
15 :- node(HSM, "s1", compositeState), not node'(NHSM, "s1", state).
16 :- node(NHSM, "s1", state), trace_node(HSM, "s1", compositeState), not node'(HSM
      , "s1", compositeState).
17 [...]
```

**Listing 1.3.** Fragment of the HSM2NHSM transformation

in line 13-14. Constraints in line 15-16 impose that each time a composite state occurs in the HSM model a correspondent state in the NHSM model is generated, and vice versa. Missing sub-states in a NHSM model can be generated again in the HSM model by means of trace information (see line 10-11 and 16). Trace elements are automatically generated each time a model element is discarded by the mapping and need to be stored in order to be regenerated during the backward transformation.

Note that the specification order of the relations is not relevant as their execution is bottom-up; i.e., the final answer set is always deduced starting from the more nested facts.

**Execution Engine.** The specified transformations are executed by a generic engine which is (partially) reported in Listing 1.4. The main goal of the transformation execution is the generation of target elements as the *node'* elements in line 11 of Listing 1.4. As previously said transformation rules may produce more than one target models, which are all the possible combinations of elements that the program is able to create. In particular, by referring to Listing 1.4 target node elements with the form node'(MM,ID,MC) are created if the following conditions are satisfied:

- the considered element is declared in the input source model. The lines 1-2 contain the rules for the source conformance checking related to node terms. In particular, the term is_source_metamodel_conform(MM,ID,MC) is true if the terms node(MM,ID,MC) and metanode(MM,MC) exist. Therefore, the term bad_source is true if the corresponding is_source_metamodel_con- form(MM,ID,MC) is valued to false with respect to the node(MM,ID,MC) source element;
- at least a relation exists between a source element and the candidate target element. In particular, the term mapping(MM,ID,MC) in line 3 is true if there exists a relation which involves elements referring to MC and MC2 metaclasses and an element node(MM2,ID,MC2). In other words, a mapping can be executed each time it is

```
1 is_source_metamodel_conform(MM,ID,MC) :- node(MM,ID,MC), metanode(MM,MC).
2 bad_source :- node(MM,ID,MC), not is_source_metamodel_conform(MM,ID,MC).
3 mapping(MM,ID,MC) :- relation(R,MM,MC), relation(R,MM2,MC2), node(MM2,ID,MC2),
      MM!=MM2.
4 is_target_metamodel_conform(MM,MC) :- metanode(MM,MC).
5 {is_generable(MM,ID,MC)} :- not bad_source, mapping(MM,ID,MC),
      is_target_metamodel_conform(MM,MC), MM=mmt.
6 node'(MM,ID,MC) :- is_generable(MM,ID,MC), mapping(MM,ID,MC), MM=mmt.
```

**Listing 1.4.** Fragment of the *Execution engine*

specified between a source and a target, and there exists the appropriate source to compute the target;
- the candidate target element conforms to the target metamodel. In particular, the term `is_target_metamodel_conform(MM,MC)` in line `6` is true if the `MC` metaclass exists in the `MM` metamodel (i.e. the target metamodel);
- finally, any constraint defined in the *relations* in Listing 1.3 is valued to false.

The invertibility of transformations is obtained by means of trace information that connect source and target elements; in this way, during the transformation process, the relationships between models that are created by the transformation executions can be stored to preserve mapping information in a permanent way. Furthermore, all the source elements lost during the forward transformation execution (for example, due to the different expressive power of the metamodels) are stored in order to be generated again in the backward transformation execution.

**Specifying Model Transformation with Janus.** Due to the reduced usability of the ASP language, we have decided to provide support for specifying transformations by means of a more human readable syntax inspired by QVT-R. In Listing 1.5 we report a fragment of the HSM2NHSM transformation specified in JTL and it transforms hierarchical state machines into flat state machines and the other way round. The forward transformation is clearly non-injective as many different hierarchical machines can be flattened to the same model and consequently transforming back a modified flat machine can give place to more than one hierarchical machine. Such a transformation consists of several relations like *StateMachine2StateMachine*, *State2State* and *CompositeState2State* which are specified in Listing 1.5. They define correspondences between *a)* state machines in the two different metamodels *b)* atomic states in the two different metamodels and *c)* composite states in hierarchical machines and atomic states in flat machines. The relation in lines 11-20 of Listing 1.5 is constrained by means of the *when* clause such that only atomic states are considered. Similarly to QVT, the *checkonly* and *enforce* constructs are also provided: the former is used to check if the domain where it is applied exists in the considered model; the latter induces the modifications of those models which do not contain the domain specified as *enforce*. A JTL relation is considered bidirectional when both the contained domains are specified with the construct *enforce*.

```
1 transformation hsm2nhsm(source : HSM, target : NHSM) {
2
3  top relation StateMachine2StateMachine {
4
5     enforce domain source sSM : HSM::StateMachine;
6     enforce domain target tSM : NHSM::StateMachine;
7
8   }
9
10 top relation State2State {
11
12    enforce domain source sourceState : HSM::State;
13    enforce domain target targetState : NHSM::State;
14
15    when {
16      sourceState.owningCompositeState.oclIsUndefined();
17    }
18
19  }
20
21 top relation CompositeState2State {
22
23   enforce domain source sourceState : HSM::CompositeState;
24   enforce domain target targetState : NHSM::State;
25
26 }
27}
```

**Listing 1.5.** A non-injective JTL program

The JTL transformations specified in the QVT-like syntax are mapped to the correspondent ASP program by means of a semantic anchoring operation as described in the next section.

**ASP Semantic Anchoring.** According to the proposed approach, the designer task is limited to specifying relational model transformations in JTL syntax and to applying them on models and metamodels defined as EMF entities within the Eclipse framework.

Designers can take advantage of ASP and of the transformation properties previously discussed in a transparent manner since only the JTL syntax is used. In fact, ASP programs are automatically obtained from JTL specifications by means of an ATL transformations as depicted in the upper part of Fig. 8. Such a transformation is able to generate ASP predicates for each relation specified with JTL. For instance, the relation *State2State* in Listing 1.5 gives place to the *relation* predicates in lines 6-7 in Listing 1.3.

The JTL *when* clause is also managed and it induces the generation of further ASP constraints. For instance, the JTL clause in line 16 of Listing 1.5 gives place to a couple of ASP constraints defined on the *owningCompositeState* feature of the state machine metamodels (see lines 8-9 in Listing 1.3). Such constraints are able to filter the states and consider only those which are not nested.

To support the backward application of the specified transformation, for each JTL relation additional ASP constraints are generated in order to support the management of trace links. For instance, the *State2State* relation in Listing 1.5 induces the generation of the constraints in lines 10-11 of Listing 1.3 to deal with the non-bijectivity of the transformation. In particular, when the transformation is backward applied on a *State* element of the target model, trace links are considered to check if such a state has

been previously generated from a source *CompositeState* or *State* element. If such trace information is missing all the possible alternatives are generated.

### 4.4 JTL in Practice

In this section we show the application of the proposed approach to the *Collapse/Expand State Diagrams* case study presented in Section 4.1. The objective is to illustrate the use of JTL in practice by exploiting the developed environment, and in particular to show how the approach is able to propagate changes dealing with non-bijective and non-total scenarios.

**Modelling State Machines.** According to the scenario described in Section 4.1, we assume that in the software development lifecycle, the designer is interested to have a behavioral description of the system by means of hierarchical state machine, whereas a test expert produces non-hierarchical state machine models. The hierarchical and non-hierarchical state machine matamodels (respectively HSM and NHSM) are given by means of their Ecore representation within the EMF framework. Then a hierarchical state machine model conforming to the HSM metamodel can be specified as the model reported in the left-hand side of Fig. 9. Models can be specified with graphical and/or concrete syntaxes depending on the tool availability for the considered modeling language. In our case, the adopted syntaxes for specifying models do not affect the overall transformation approach since models are manipulated by considering their abstract syntaxes.

**Specifying and Applying the HSM2NHSM Model Transformation.** Starting from the definition of the involved metamodels, the JTL transformation is specified according to the QVT-like syntax described in Section 4.3 (see Listing 1.5). By referring to Fig. 8, the *JTL program*, the *source* and *target metamodel*s and the *source model* have been created and need to be translated in their ASP encoding in order to be executed from the transformation engine. The corresponding ASP encodings are automatically produced by the mechanism illustrated in Section 4.3. In particular, the ASP encoding of both source model and source and target metamodels is generated according to the Listing 1.2 and 1.1, while the JTL program is translated to the corresponding ASP program (see Listing 1.3).

After this phase, the application of the HSM2NHSM transformation on *sampleHSM* generates the corresponding *sampleNHSM* model as depicted in the right part of Fig. 8. Note that, by re-applying the transformation in the backward direction it is possible to obtain again the *sampleHSM* source model. The missing sub-states and the transitions involving them are restored by means of trace information.

**Propagating Changes.** Suppose that in a refinement step the designer needs to manually modify the generated target by the changes described in Section 4.1 (see $\Delta$ changes depicted in Fig. 6), that is:

1. renaming the initial state from `Begin Installation` to `Start Install shield`;

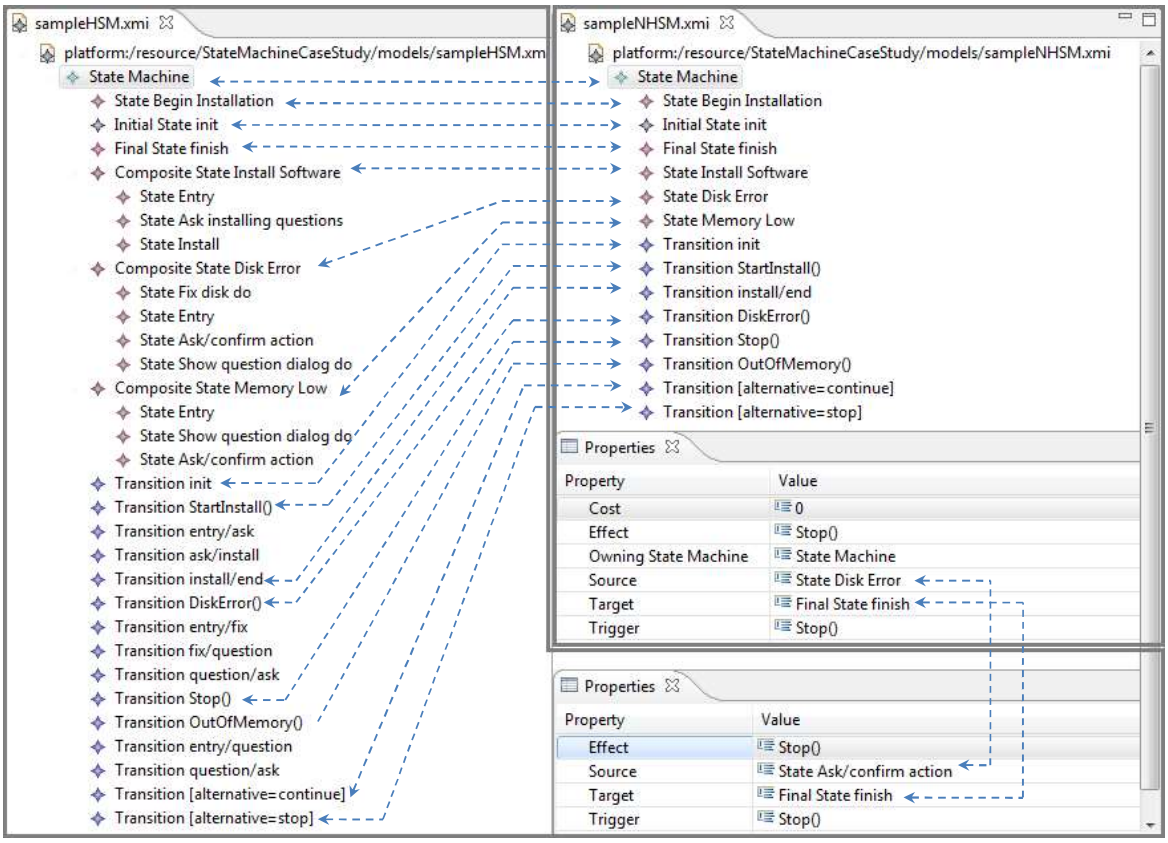



**Fig. 9.** HSM source model and the correspondent NHSM target model

2. adding the alternative `try again` to the state `Disk Error` to come back to `Install software`;
3. changing the attributes related to memory requirements (`m=500`) in the state `Install software` and cost (`c=200`) of the transition from `Memory low` to `Install software`.

The target model including such changes (*sampleNHSM'*) is shown in the left part of the Fig. 10. If the transformation HSM2NHSM is applied on it, we expect changes to be propagated back to the source model. However, due to the different expressive power of the involved metamodels, target changes may be propagated in a number of different ways, thus making the application of the reverse transformation to propose more solutions. The generated sources, namely *sampleHSM'_1/2/3/4* can be inspected through Figure 10: the change (1) has been propagated renaming the state to `Start Install shield`; the change (2) gives place to a non-bijective mapping and for this reason more than one model is generated. As previously said, the new transition can be equally targeted to each one of the nested states within `Install Software` as well as to the super state itself (see the properties *sampleHSM'_1/2/3/4* in Figure 10). For example, as visible in the property of the transition, *sampleHSM'_1* represents the case in which the transition is targeted to the composite state `Install Software`; finally, the change (3) is out of the domain of the transformation. In this case, the new values for memory and cost are not propagated on the generated source models.

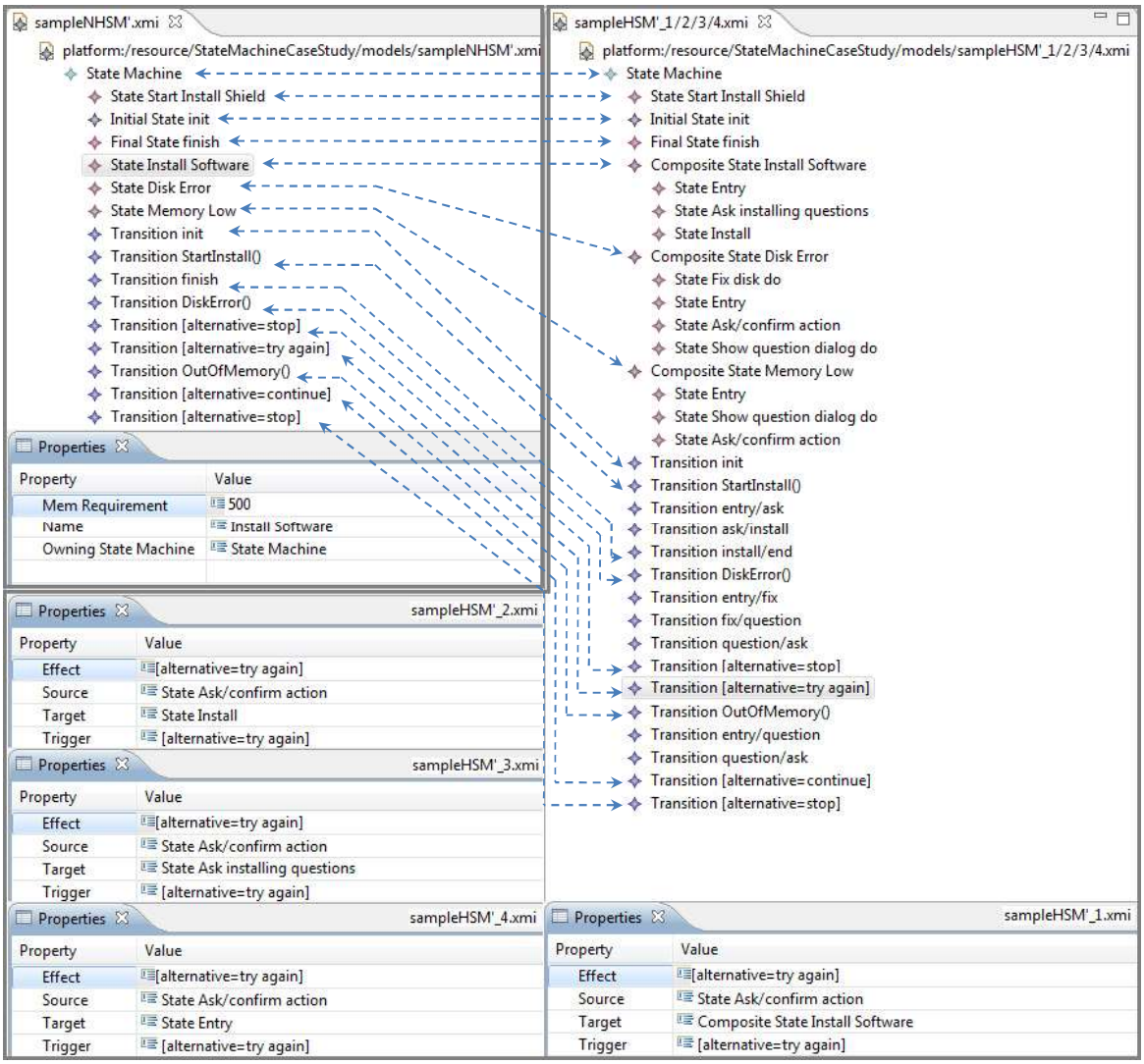



**Fig. 10.** The modified NHSM target model and the correspondent HSM source models

Even in this case, if the transformation is applied on one of the derived *sampleHSM'* models, the appropriate *sampleNHSM'* models including all the changes are generated. However, this time the target will preserve information about the chosen *sampleHSM'* source model, thus causing future applications of the backward transformation to generate only *sampleHSM'*.

With regard to the performances of our approach, we performed no formal study on its complexity yet, since that goes beyond the scope of this work; however, our observations showed that the time required to execute each transformation in the illustrated case study is more than acceptable since it always took less than one second. In the general case, when there are a lot of target alternative models the overall performance of the approach may degrade.

## 5 Application Scenario 2: Metamodel/Model Coupled Evolution

Metamodels can be considered one of the constituting concepts of MDE, since they are the formal definition of well-formed models, or in other words they constitute the

languages by which a given reality can be described in some abstract sense [2]. Metamodels are expected to evolve during their life-cycle, thus causing possible problems to existing models which conform to the old version of the metamodel and do not conform to the new version anymore. The problem is due to the incompatibility between the metamodel revisions and a possible solution is the adoption of mechanisms of model co-evolution, i.e. models need to be migrated in new instances according to the changes of the corresponding metamodel. Unfortunately, model co-evolution is not always simple and presents intrinsic difficulties which are related to the kind of evolution the metamodel has been subject to. Going into more details, metamodels may evolve in different ways: some changes may be additive and independent from the other elements, thus requiring no or little instance revision. In other cases metamodel manipulations introduce incompatibilities and inconsistencies which can not be easily (and automatically) resolved.

This section proposes an approach based on higher-order model transformations (HOTs) to model coupled evolution [65]. In particular, HOTs take a difference model formalizing the metamodel modifications and generate a model transformation able to adapt and recovery the validity of the compromised models. The approach has been applied successfully in different application domains, e.g., to manage the evolution of Web applications [66].

### 5.1 Metamodel Differences

In Fig. 11 it is depicted an example of the evolution of a (simplified) Petri Net metamodel, which takes inspiration from the work in [18]. The initial Petri Net ($\texttt{MM}_0$) consists of `Places` and `Transitions`; moreover, places can have source and/or destination transitions, whereas transitions must link source and destination places (`src` and `dst` association roles, respectively). In the new metamodel $\texttt{MM}_1$, each `Net` has at least one `Place` and one `Transition`. Besides, arcs between places and transitions are made explicit by extracting `PTArc` and `TPArc` metaclasses. This refinement permits to add further properties to relationships between places and transitions. For example, the Petri Net formalism can be extended by annotating arcs with weights. As `PTArc` and `TPArc` both represent arcs, they can be generalized by a superclass, and a new integer metaproperty can be added in it. Therefore, an abstract class `Arc` encompassing the integer metaproperty `weight` has been added in $\texttt{MM}_2$ revision of the metamodel. Finally, `Net` has been renamed into `PetriNet`. The metamodels in Fig. 11 will be exploited as the running example throughout this section. They have been kept deliberately simple because of space limitations, even though they are suitable to present all the insights of the co-adaptation mechanisms as already demonstrated in [18].

The revisions illustrated so far can invalidate existing instances; therefore, each version needs to be analysed to comprehend the various kind of updates it has been subject to and, eventually, to elicit the necessary adaptations of corresponding models. Metamodel manipulations can be classified by their corrupting or non-corrupting effects on existing instances [67]:

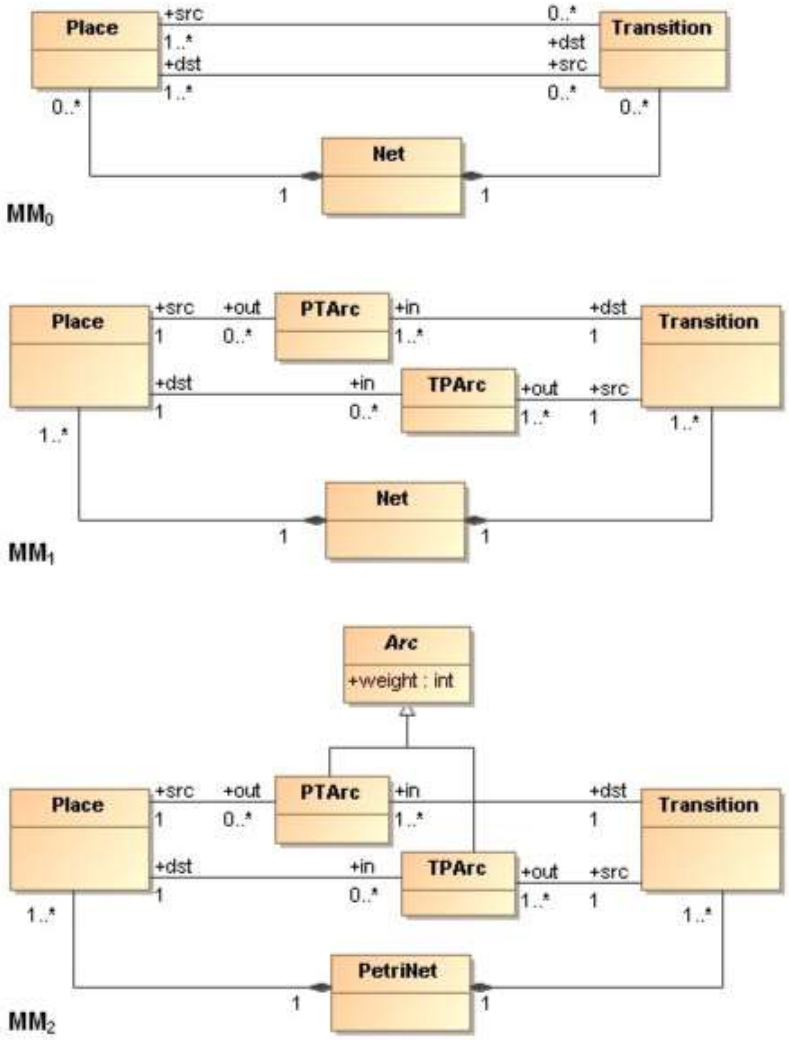



**Fig. 11.** Petri Net metamodel evolution

- *non-breaking changes*: changes which do not break the conformance of models to the corresponding metamodel;
- *breaking and resolvable changes*: changes which break the conformance of models even though they can be automatically co-adapted;
- *breaking and unresolvable changes*: changes which break the conformance of models which can not automatically co-evolved and user intervention is required.

In other words, *non-breaking changes* consist of additions of new elements in a metamodel `MM` leading to `MM′` without compromising models which conform to `MM` and thus, in turn, conform to `MM′`. For instance, in the metamodel $\texttt{MM}_2$ illustrated in Fig. 11 the abstract metaclass `Arc` has been added as a generalization of the `PTArc` and `TPArc` metaclasses (without considering the new attribute `weight`). After such a modification, models conforming to $\texttt{MM}_1$ still conform to $\texttt{MM}_2$ and co-evolution is not necessary. Unfortunately, this is not always the case since in general changes may break models even though sometimes automatic resolution can be performed, i.e. when facing *breaking and resolvable changes*. For instance, the Petri Net metamodel $\texttt{MM}_1$ in Fig. 11 is enriched with the new `PTArc` and `TPArc` metaclasses. Such a modification breaks the models that conform to $\texttt{MM}_0$ since according to the new metamodel $\texttt{MM}_1$, `Place` and

`Transition` instances can not be directly related, but `PTArc` and `TPArc` elements are required. However, models can be automatically migrated by adding for each couple of `Place` and `Transition` entities two additional `PTArc` and `TPArc` instances between them.

Often manual interventions are needed to solve breaking changes like, for instance, the addition of the new attribute `weight` to the class `Arc` of $\texttt{MM}_2$ in Fig. 11 which were not specified in $\texttt{MM}_1$. The models conforming to $\texttt{MM}_1$ can not be automatically co-evolved since only a human intervention can introduce the missing information related to the weight of the arc being specified, or otherwise default values have to be considered. We refer to such situations as *breaking and unresolvable changes*.

All the scenarios of model co-adaptations can be managed with respect to the possible metamodel modifications which can be distinguished into *additive*, *subtractive*, and *updative*. In particular, with additive changes we refer to metamodel element additions which in turn can be further distinguished as follows:

- *Add metaclass*: introducing new metaclasses is a common practice in metamodel evolution which gives place to metamodel extensions. Adding new metaclasses raises co-evolution issues only if the new elements are mandatory with respect to the specified cardinality. In this case, new instances of the added metaclass have to be accordingly introduced in the existing models;
- *Add metaproperty*: this is similar to the previous case since a new metaproperty may be or not obligatory with respect to the specified cardinality. The existing models maintain the conformance to the considered metamodel if the addition occurs in abstract metaclasses without subclasses; in other cases, human intervention is required to specify the value of the added property in all the involved model elements;
- *Generalize metaproperty*: a metaproperty is generalized when its multiplicity or type are relaxed. For instance, if the cardinality `3..n` of a sample metaclass `MC` is modified in `0..n`, no co-evolution actions are required on the corresponding models since the existing instances of `MC` still conform to the new version of the metaclass;
- *Pull metaproperty*: a metaproperty `p` is pulled in a superclass `A` and the old one is removed from a subclass `B`. As a consequence, the instances of the metaclass `A` have to be modified by inheriting the value of `p` from the instances of the metaclass `B`;
- *Extract superclass*: a superclass is extracted in a hierarchy and a set of properties is pulled on. If the superclass is abstract model instances are preserved, otherwise the effects are referable to metaproperty pulls.

Subtractive changes consist of the deletions of some of the existing metamodel elements as described in the following:

- *Eliminate metaclass*: a metaclass is deleted by giving place to a sub metamodel of the initial one. In general, such a change induces in the corresponding models the deletions of all the metaclass instances. Moreover, if the involved metaclass has subclasses or it is referred by other metaclasses, the elimitation causes side effects also to the related entities;

**Table 1.** Changes classification

| Change type | Change |
|---|---|
| **Non-breaking changes** | Generalize metaproperty |
| | Add (non-obligatory) metaclass |
| | Add (non-obligatory) metaproperty |
| **Breaking and resolvable changes** | Extract (abstract) superclass |
| | Eliminate metaclass |
| | Eliminate metaproperty |
| | Push metaproperty |
| | Flatten hierarchy |
| | Rename metaelement |
| | Move metaproperty |
| | Extract/inline metaclass |
| **Breaking and unresolvable changes** | Add obligatory metaclass |
| | Add obligatory metaproperty |
| | Pull metaproperty |
| | Restrict metaproperty |
| | Extract (non-abstract) superclass |

- *Eliminate metaproperty*: a property is eliminated from a metaclass, it has the same effect of the previous modification;
- *Push metaproperty*: pushing a property in subclasses means that it is deleted from an initial superclass `A` and then cloned in all the subclasses `C` of `A`. If `A` is abstract then such a metamodel modification does not require any model co-adaptation, otherwise all the instances of `A` and its subclasses need to be accordingly modified;
- *Flatten hierarchy*: to flatten a hierarchy means eliminating a superclass and introducing all its properties into the subclasses. This scenario can be referred to metaproperty pushes;
- *Restrict metaproperty*: a metaproperty is restricted when its multiplicity or type are enforced. It is a complex case where instances need to be co-adapted or restricted. Restricting the upper bound of the multiplicity requires a selection of certain values to be deleted. Increasing the lower bound requires new values to be added for the involved element which usually are manually provided. Restricting the type of a property requires type conversion for each value.

Finally, a new version of the model can consist of some updates of already existing elements leading to updative modifications which can be grouped as follows:

- *Rename metaelement*: renaming is a simple case in which the change needs to be propagated to existing instances and can be performed in an automatic way;
- *Move metaproperty*: it consists of moving a property `p` from a metaclass `A` to a metaclass `B`. This is a resolvable change and the existing models can be easily co-evolved by moving the property `p` from all the instances of the metaclass `A` to the instances of `B`;

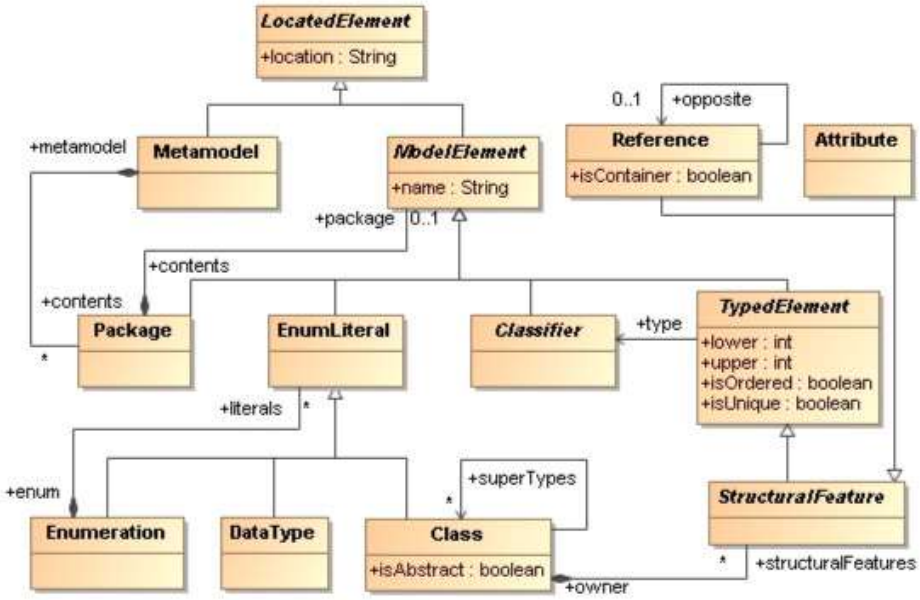



**Fig. 12.** KM3 metamodel

– *Extract/inline metaclass*: extracting a metaclass means to create a new class and move the relevant fields from the old class into the new one. Vice versa, to inline a metaclass means to move all its features into another class and delete the former. Both metamodel refactorings induce automated model co-evolutions.

The classification illustrated so far is summarized in Tab. 1 and makes evident the fundamental role of evolution representation. At a first glance it seems that the classification does not encompass *references* that are associations amongst metaclasses. However, references can be considered properties of metaclasses at the same level of attributes.

Metamodel evolutions can be precisely categorized by understanding the kind of modifications a metamodel undergone. Moreover, starting from the classification it is possible to adopt adequate countermeasures to co-evolve existing instances. Nonetheless, it is worth noting that the classification summarized in Tab. 1 is based on a clear distinction between the metamodel evolution categories. Unfortunately, in real world experiences the evolution of a metamodel can not be reduced to a sequence of atomic changes, generally several types of changes are operated as affecting multiple elements with different impacts on the co-adaptation. Furthermore, the entities involved in the evolution can be related one another. Therefore, since co-adaptation mechanisms are based on the described change classification, a metamodel adaptation will need to be decomposed in terms of the induced co-evolution categories. The possibility to have a set of dependences among the several parts of the evolution makes the updates not always distinguishable as single atomic steps of the metamodel revision, but requires a further refinement of the classification as introduced in the next section and discussed in details in Sect. 5.3.

### 5.2 Formalizing Metamodel Differences

The problem of model differences is intrinsically complex and requires specialized algorithms and notations to match the abstraction level of models [68]. Recently, in [69, 70] two similar techniques have been introduced to represent differences as models, hereafter called *difference models*; interestingly these proposals combine the advantages of declarative difference representations and enable the reconstruction of the final model by means of automated transformations which are inherently defined in the approaches. In the rest of the section, we recall the difference representation approach defined in [69] in order to provide the reader with the technical details which underpin the solution proposed in Sect. 5.3.

Despite the work in [69] has been introduced to deal with model revisions, it is easily adaptable to metamodel evolutions too. In fact, a metamodel is a model itself, which conforms to a metamodel referred to as the meta metamodel [2]. For presentation purposes, the KM3 language in Fig. 12 is considered throughout the paper, even though the solution can be generalized to any metamodeling language like OMG/MOF [23] or EMF/Ecore [29].

The overall structure of the change representation mechanism is depicted in Fig. 13: given two *base metamodels* $\texttt{MM}_1$ and $\texttt{MM}_2$ which conform to an arbitrary *base meta metamodel* (KM3 in our case), their difference conforms to a *difference metamodel* `MMD` derived from KM3 by means of an automated transformation `MM2MMD`. The base meta metamodel, extended as prescribed by such a transformation, consists of new constructs able to represent the possible modifications that can occur on metamodels and which can be grouped as follows:

- *additions*: new elements are added in the initial metamodel; with respect to the classification given in Sect. 5.1, *Add metaclass* and *Extract superclass* involve this kind of change;
- *deletions*: some of the existing elements are deleted as a whole. *Eliminate metaclass* and *Flatten hierarchy* fall in this category of manipulations;
- *changes*: a new version of the metamodel being considered can consist of updates of already existing elements. For instance, *Rename metaelement* and *Restrict metaproperty* require this type of modification. Also the addition and deletion of metaproperty (i.e. *Add metaproperty* and *Eliminate metaproperty*, respectively) are modelled through this construct. In fact, when a metaelement is included in a container the manipulation is represented as a *change* of the container itself.

In order to represent the differences between the Petri Net metamodel revisions, the extended KM3 meta metamodel depicted in Fig. 14 is generated by applying the `MM2MMD` transformation in Fig. 13 previously mentioned. For each metaclass `MC` of the KM3 metamodel, the additional metaclasses `AddedMC`, `DeletedMC`, and `ChangedMC` are generated. For instance, the metaclass `Class` in Fig. 12 induces the generation of the metaclasses `AddedClass`, `DeletedClass`, and `ChangedClass` as depicted in Fig. 14. In the same way, `Reference` metaclass induces the generation of the metaclasses `AddedReference`, `DeletedReference`, and `ChangedReference`.

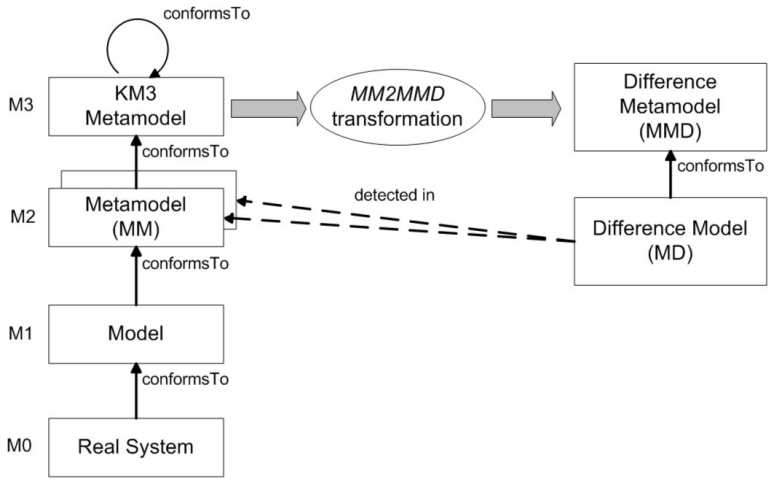



**Fig. 13.** Overall structure of the model difference representation approach

The generated difference metamodel is able to represent all the differences amongst metamodels which conform to KM3. For instance, the model in Fig. 15 conforms to the generated metamodel in Fig. 14 and represents the differences between the Petri Net metamodels specified in Fig. 11. The differences depicted in such a model can be summarized as follows:

1) the addition of the new class `PTArc` in the $\mathtt{MM}_1$ revision of the Petri Net metamodel is represented by means of an `AddedClass` instance, as illustrated by model difference $\Delta_{0,1}$ in Fig. 15. Moreover, the reference between `Place` and `Transition` named `dst` has been updated to link `PTArc` with name `out`. Analogously, the reverse reference named `src` has been manipulated to point `PTArc` and named as `in`. Two new references have been added through the corresponding `AddedReference` instances to realize the reverse links from `PTArc` to `Place` and `Transition`, respectively. Finally, the composition relationship between `Net` and `Place` has been updated by prescribing the existence of at least one `Place` through the `lower` property which has been updated from `0` to `1`. The same enforcement has been done to the composition between `Net` and `Transition`;
2) the addition of the new abstract class `Arc` in $\mathtt{MM}_2$ together with its attribute `weight` is represented through an instance of the `AddedClass` and the `AddedAttribute` metaclasses in the $\Delta_{1,2}$ delta of Fig. 15. In the meanwhile, `PTArc` and `TPArc` classes are made specializations of `Arc`. Finally, `Net` entity is renamed as `PetriNet`.

Difference models like the one in Fig. 15 can be obtained by using today's available tools like EMFCompare [71] and SiDiff [72].

The representation mechanism used so far allows to identify changes which occurred in a metamodel revision and satisfies a number of properties, as illustrated in [69]. One of them is the *compositionality*, i.e. the possibility to combine difference models in interesting constructions like the sequential and the parallel compositions, which in turn result in valid difference models themselves. For the sake of simplicity, let us consider only two modifications over the initial model: the sequential composition of such

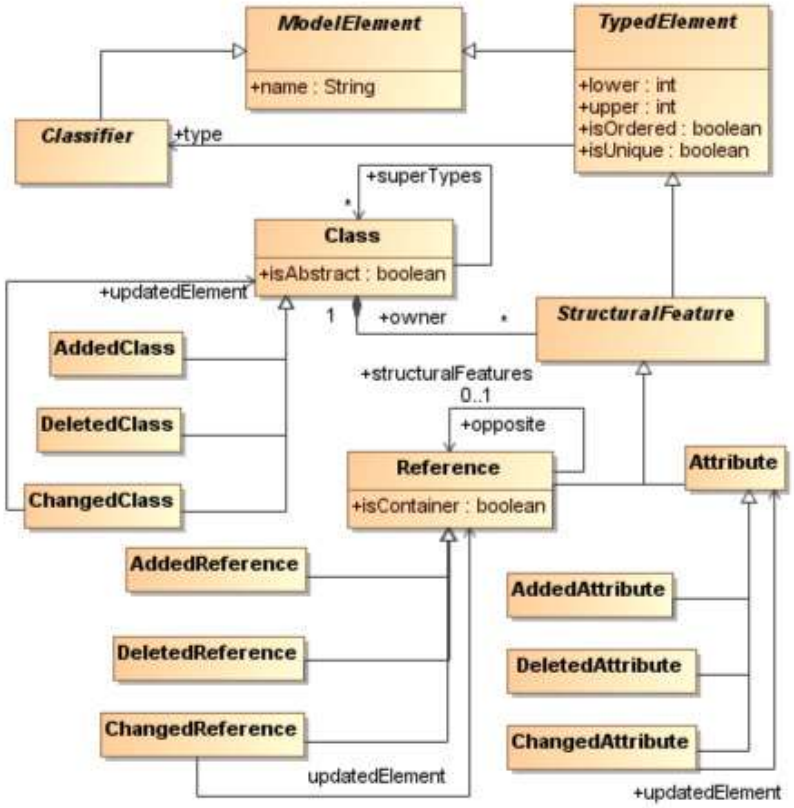



**Fig. 14.** Generated difference KM3 metamodel

manipulations corresponds to merging the modifications conveyed by the first document and then, in turn, by the second one in a resulting difference model containing a minimal difference set, i.e., only those modifications which have not been overridden by subsequent manipulations. Whereas, parallel compositions are exploited to combine modifications operated from the same ancestor in a concurrent way. In case both manipulations are not affecting the same elements they are said *parallel independent* and their composition is obtained by merging the difference models by interleaving the single changes and assimilating it to the sequential composition. Otherwise, they are referred to as *parallel dependent* and conflict issues can arise which need to be detected and resolved [73].

Finally, difference documentation can be exploited to re-apply changes to arbitrary input models (see [69] for further details) and for managing model co-evolution induced by metamodel manipulations. In the latter case, once differences between metamodel versions have been detected and represented, they have to be partitioned in resolvable and non resolvable scenarios in order to adopt the corresponding resolution strategy. However, this distinction is not always feasible because of parallel dependent changes, i.e. situations where multiple changes are mixed and interdependent one another, like when a resolvable change is in some way related with a non-resolvable one, for instance. In those cases, deltas have to be decomposed in order to isolate the non-resolvable portion from the resolvable one, as illustrated in the next section.

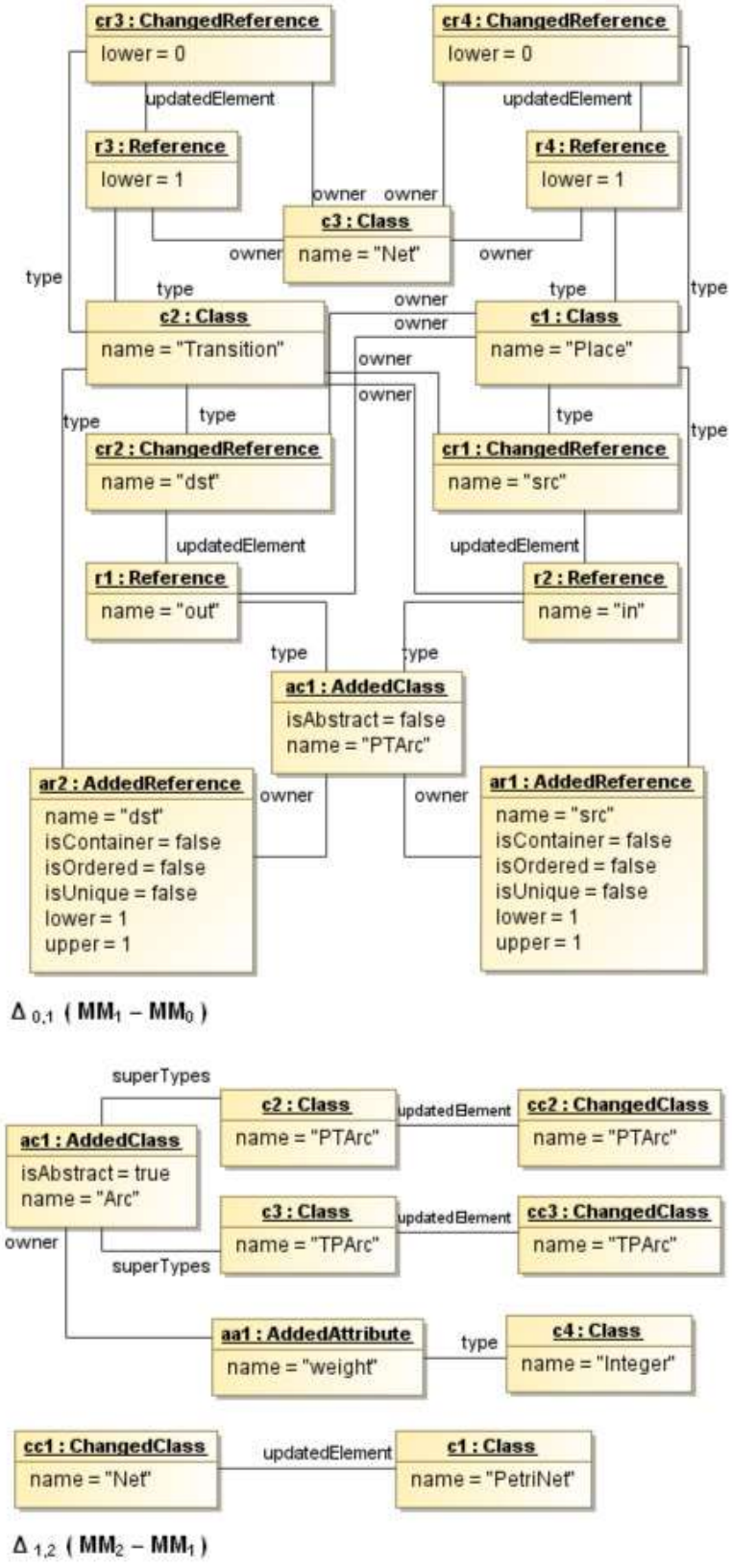



**Fig. 15.** Subsequent Petri Net metamodel adaptations

### 5.3 Transformational Adaptation of Models

This section proposes a transformational approach able to consistently adapt existing models with respect to the modifications occurred in the corresponding metamodels. The proposal is based on model transformation and the difference representation techniques presented in the previous section. In particular, given two versions $\texttt{MM}_1$ and $\texttt{MM}_2$ of the same metamodel (see Fig. 16.a), their differences are recorded in a difference model $\Delta$, whose metamodel `KM3Diff` is automatically derived from `KM3` as described in Sect. 5.2. In realistic cases, the modifications consist of an arbitrary combination of the atomic changes summarized in Tab. 1. Hence, a difference model formalizes all kind of modifications, i.e. non-breaking, breaking resolvable and unresolvable ones. This poses additional difficulties since current approaches (e.g. [18, 67]) do not provide any support to co-adaptation when the modifications are given without explicitly distinguishing among breaking resolvable and unresolvable changes. Our approach consists of the following steps:

*i)* automatic decomposition of $\Delta$ in two disjoint (sub) models, $\Delta_R$ and $\Delta_{\neg R}$, which denote breaking resolvable and unresolvable changes;
*ii)* if $\Delta_R$ and $\Delta_{\neg R}$ are *parallel independent* (see previous section) then we separately generate the corresponding co-evolutions;
*iii)* if $\Delta_R$ and $\Delta_{\neg R}$ are *parallel dependent*, they are further refined to identify and isolate the interdependencies causing the interferences.

The distinction between *ii)* and *iii)* is due to fact that when two modifications are not independent their effects depend on the order the changes occur leading to non confluent situations. The confluence can still be obtained by removing those modifications which caused the conflicts as described in Sect. 5.3.

The general approach is outlined in Figure 16 where dotted and solid arrows represent conformance and transformation relations, respectively, and square boxes are any kind of models, i.e. models, difference models, metamodels, and even transformations. In particular, the decomposition of $\Delta$ is given by two model transformations, $T_R$ and $T_{\neg R}$ (right-hand side of Fig. 16.a). Co-evolution actions are directly obtained as model transformations from metamodel changes by means of higher-order transformations, i.e. transformations which produce other transformations [2]. More specifically, the higher-order transformations $\mathcal{H}_R$ and $\mathcal{H}_{\neg R}$ (see Fig. 16.b and 16.c) take $\Delta_R$ and $\Delta_{\neg R}$ and produce the (co-evolving) model transformations $CT_R$ and $CT_{\neg R}$, respectively. Since $\Delta_R$ and $\Delta_{\neg R}$ are parallel independent $CT_R$ and $CT_{\neg R}$ can be applied in any order because they operate to disjoint sets of model elements, or in other words

$$(CT_{\neg R} \cdot CT_R)(M_1) = (CT_R \cdot CT_{\neg R})(M_1) = M_2$$

with $M_1$ and $M_2$ models conforming to the metamodel $\texttt{MM}_1$ and $\texttt{MM}_2$, respectively (see Fig. 16.d).

In the rest of the section we illustrate the approach and its implementation. In particular, we describe the decomposition of $\Delta$ and the generation of the co-evolving model transformations for the case of parallel independent breaking resolvable and

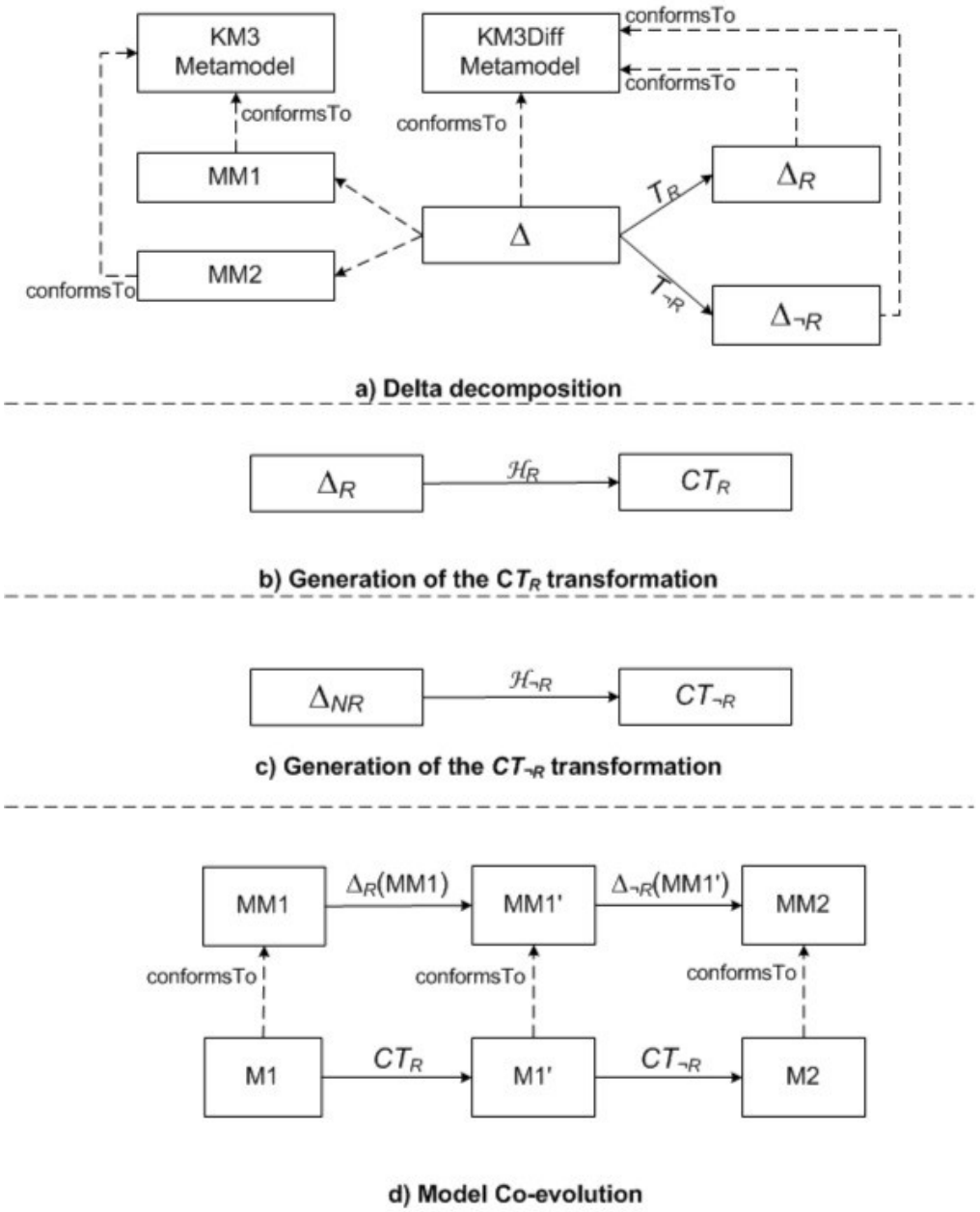



**Fig. 16.** Overall approach

unresolvable changes. Finally, we outline how to remove interdependencies from parallel dependent changes in order to generalize the solution provided in Sect. 5.3.

**Parallel Independent Changes.** The generation of the co-evolving model transformations is described in the rest of the section by means of the evolutions the `PetriNet` metamodel has been subject to in Fig. 11. The differences between the subsequent metamodel versions are given in Fig. 15 and have, in turn, to be decomposed to distinguish breaking resolvable and unresolvable modifications.

In particular, the difference $\Delta_{(0,1)}$ from $\texttt{MM}_0$ to $\texttt{MM}_1$ consists of two atomic modifications, i.e. an *Extract metaclass* and a *Restrict metaproperty* change (according to the classification in Tab. 1), which are referring to different sets of model elements. The approach is able to detect parallel independence by verifying that the eventual decomposed differences have an empty intersection. Since *a)* the previous atomic changes are breaking resolvable and unresolvable, and *b)* they do not share any model element, then $\Delta_{(0,1)}$ is decomposed by $T_R$ and $T_{\neg R}$ into the parallel independent $\Delta_{R(0,1)}$ and $\Delta_{\neg R(0,1)}$, respectively. In fact, the former contains the extract metaclass action which affects the elements `Place` and `Transition`, whereas the latter holds the restrict metaproperty changes consisting of the reference modifications in the metaclass `Net`. Analogously, the same decomposition can be operated on $\Delta_{(1,2)}$ (denoting the evolution from $\texttt{MM}_1$ to $\texttt{MM}_2$) to obtain $\Delta_{R(1,2)}$ and $\Delta_{\neg R(1,2)}$ since the denoted modifications do not conflict one another. In fact, the *Rename metaelement* change (represented by `cc1` and `c1` in Fig. 15.b) is applied to `Net`, whereas the *Add obligatory metaproperty* operation involves the new metaclass `Arc` which is supertype of the `PTArc` and `TPArc` metaclasses.

As previously said, once the $\Delta$ is decomposed the higher-order transformations $\mathcal{H}_R$ and $\mathcal{H}_{\neg R}$ detect the occurred metamodel changes and accordingly generate the coevolution to adapt the corresponding models. In the current implementation, model transformations are given in ATL, a QVT compliant language part of the AMMA platform [64] which contains a mixture of declarative and imperative constructs. In the Listing 1.6 a fragment of the $\mathcal{H}_R$ transformation is reported: it consists of a module specification containing a header section (lines 1-2), transformation rules (lines 4-41) and a number of helpers which are used to navigate models and to define complex calculations on them. In particular, the header specifies the source models, the corresponding metamodels, and the target ones. Since the $\mathcal{H}_R$ transformation is higher-order, the target model conforms to the ATL metamodel which essentially specifies the abstract syntax of the transformation language. Moreover, $\mathcal{H}_R$ takes as input the model which represents the metamodel differences conforming to `KM3Diff`.

The helpers and the rules are the constructs used to specify the transformation behaviour. The source pattern of the rules (e.g. lines 15-20) consists of a source type and a OCL [51] guard stating the elements to be matched. Each rule specifies a target pattern (e.g. lines 21-25) which is composed of a set of elements, each of them (as the one at lines 22-25) specifies a target type from the target metamodel (for instance, the type `MatchedRule` from the ATL metamodel) and a set of bindings. A binding refers to a feature of the type, i.e. an attribute, a reference or an association end, and specifies an expression whose value initializes the feature. $\mathcal{H}_R$ consists of a set of rules each of them devoted to the management of one of the resolvable metamodel changes reported in Tab. 1. For instance, the Listing 1.6 contains the rules for generating the coevolution actions corresponding to the *Rename metaelement* and the *Extract metaclass* changes.

```
1 module H_R;
2 create  OUT : ATL  from Delta : KM3Diff;
3 ...
4 rule atlModule {
5     from
6        s: KM3Diff!Metamodel
7     to
8      t : ATL!Module (
9           name <- 'CTR',
10          outModels <- Sequence {tm},
11       inModels <- Sequence {sm},...
12     ),...
13 }
14 rule CreateRenaming {
15   from
16      input : KM3Diff!Class,
17      delta : KM3Diff!ChangedClass
18      ...
19      (not input.isAbstract
20       and input.name <> delta.updatedElement.name...)
21   to
22      matchedRule :  ATL!MatchedRule (
23        name<-input.name + '2' + delta.updatedElement.name,
24        ...
25      ),...
26 }
27 rule CreateExtractMetaClass {
28  from
29      cr1: KM3Diff!ChangedReference, cr2: KM3Diff!ChangedReference, r1 : KM3Diff!
            Reference, r2 : KM3Diff!Reference, c1 : KM3Diff!Class,
30      c2 : KM3Diff!Class,...
31      ( cr1.updatedElement = r2 and cr1.owner = c2
32        and cr1.type = c1 and ...)
33  to
34      -- MatchedRule generation
35      matchedRule_i_c2 :  ATL!MatchedRule (
36         name<-i_c2.name + '2' + i_c2.name,
37         inPattern <- ip_i_c2,
38         outPattern <- op_i_c2,
39         ...
40      ),...
41 }
42 ...
```

**Listing 1.6.** Fragment of the $HOT_R$ transformation

The application of $\mathcal{H}_R$ to the metamodel $\texttt{MM}_0$ in Fig. 11 and the difference model $\Delta_{R(0,1)}$ in Fig. 15 generates the model transformation reported in the Listing 1.7. In fact, the source pattern of the `CreateExtractMetaClass` rule (lines 28-32 in the Listing 1.6) matches with the two *Extract metaclass* changes represented in $\Delta_{R(0,1)}$. They consist of the additions of the `PTArc` and `TPArc` metaclasses instead of the direct references between the existing elements `Place` and `Transition`. Consequently, according to the structural features of the involved elements, the `CreateExtractMetaClass` rule generates the transformation $CT_{R(0,1)}$ which is able to co-evolve all the models conforming to $\texttt{MM}_0$ by adapting them with respect to the new metamodel $\texttt{MM}_1$ (see line 1-2 of the Listing 1.7). In particular, each element of type `Place` has to be modified by changing all the references to elements of type `Transition` with references to new elements of type `PTArc` (see lines 4-23 in the Listing 1.7). The same modification has to be performed for all the elements of type `Transition` by creating new elements of type `TPArc` which have to be added instead of direct references between `Transition` and `Place` instances (see lines 24-42).

```
module CTR;
create OUT : MM1 from IN : MM0;
...
rule Place2Place {
  from
    s : MM1!Place
    ...
  to
    t : MM2!Place (
      name <- s.name,
      net <- s.net,
      out <- s.dst->collect(e |
        thisModule.createPTArc(e, t)
      )
    )
}
rule createPTArc(s : OclAny, n : OclAny) {
  to
    t : MM2!PTArc (
      src <- s,
      dst <- n
    ), ...
}
rule Transition2Transition {
  from
    s : MM1!Transition
    ...
  to
    t : MM2!Transition (
      net <- s.net,
      in <- s.dst->collect(e |
        thisModule.createTPArc(e, t)
      )
    )
}
rule createTPArc(s : OclAny, n : OclAny) {
  to
    t : MM2!PTArc (
      dst <- s,
      src <- n
    ), ...
}
...
```

**Listing 1.7.** Fragment of the generated $CT_{R(0,1)}$ transformation

The management of the breaking and unresolvable modifications is based on the same techniques presented so far for the breaking resolvable case. However, as mentioned in Sect. 5.1, the involved transformations can not automatically co-adapt the models but are limited to default actions which have to be refined by the designer.

**Parallel Dependent Changes.** As mentioned above, the automatic co-adaptation of models relies on the parallel independence of breaking resolvable and unresolvable modifications, or more formally

$$\Delta_R | \Delta_{\neg R} = \Delta_R ; \Delta_{\neg R} + \Delta_{\neg R} ; \Delta_R \tag{1}$$

where $+$ denotes the non-deterministic choice. In essence, their application is not affected by the adopted order since they do not present any interdependencies. In case the modifications in Tab. 1 refer to the same elements then the order in which such modifications take place matters and does not allow the decomposition of a difference model

as, for instance, when evolving $\mathtt{MM}_0$ directly to $\mathtt{MM}_2$ (although the sub steps $\mathtt{MM}_0 - \mathtt{MM}_1$ and $\mathtt{MM}_1 - \mathtt{MM}_2$ are directly manageable as described in the previous section).

A possible approach, which is only sketched in the following, consists in isolating the interdependencies whenever (1) does not hold. The intention is to define an iterative process consisting in *diminishing* the modifications between two metamodels until the corresponding breaking resolvable and unresolvable differences are parallel independent. In particular, let $\Delta$ be a difference between two metamodels, then we denote by $\mathcal{P}(\Delta)$ the *difference powermodel*, that is the (partially ordered) set of all possible valid sub models of $\Delta$ (i.e. fragments of the difference model which are still conforming to the difference metamodel)

$$\mathcal{P}(\Delta) = \{\delta_0 = \phi, \cdots, \delta_i, \delta_{i+1}, \cdots, \delta_n = \Delta\}$$

Then, the solution is the smallest $k$ in $\{0, \cdots, n\}$ such that

$$\Delta^{(k)}; \delta_k = \Delta$$

where $\Delta^{(k)}$ is the difference model between $\Delta$ and $\delta_k$, and

$$\Delta^{(k)} = \Delta_R^{(k)} | \Delta_{\neg R}^{(k)}$$

with $\Delta_R^{(k)}$ and $\Delta_{\neg R}^{(k)}$ parallel independent. Hence, the problem of parallel dependence is reduced to the following

$$\Delta = (\Delta_R^{(k)} | \Delta_{\neg R}^{(k)}); \delta_k$$

by applying the higher-order transformation introduced in the previous section. For instance, if we consider $(\mathtt{MM}_2 - \mathtt{MM}_0)$ the solution consists in iteratively finding a difference model which maps $\mathtt{MM}_0$ to the intermediate metamodel corresponding to $\mathtt{MM}_2$ without the attribute *weight* of the `Arc` metaclass. Therefore, the remaining $\delta_k$ in this example is a non resolvable change, while in general it may demand for further iterations of the decomposition process.

The problem of finding the correct scheduling of the adaptation steps has been solved in [74] which proposes a dependency analysis which underpins a resolution strategy for their correct application. In particular, all the metamodel change dependencies have been considered and for each of them a resolution schema is proposed enabling the complete automation of the adaptation. Interestingly, the technique is independent from the metamodel and its underlying semantics, since it relies only on the definition of the metamodeling language.

## 6 Conclusions

In this paper, model transformation approaches have been illustrated. They have been grouped according to (macro) characteristics which distinguished their intrinsic features. A number of languages which are prominent in their specificity have been briefly discussed. Finally, two different application scenarios have been presented in order to

illustrate complex situations where model transformations have been successfully applied. In the first case, the JTL language has been presented and particularly its capability in dealing with non-bijective transformations which present interesting difficulties when modifications operated on a target model must be back propagated to source models. The second case illustrated an application of higher-order transformations to the problem of the coupled evolution of metamodels and models. In partcicular, the evolution of a metamodel is specified in a difference model that once entered in a given HOT produces other transformations capable of adapting those models which have been invalidated by the metamodel changes.

Model transformations are considered among the most distinguished element of MDE as their constitute the main gluing and composing mechanism within any MDE ecosystem. However, the maturity of this field is still to be assessed as many aspects still need to be further investigated. In general, very important aspects such as bidirectionality and change propagation have been already object of intense debate, as witnessed by the work in [57], while genericity [75] and model typing [76] has been only more recently considered. Other aspects, like transformation semantics, strategies and tools for testing and verifying transformations, are addressed in another course [77] of the SFM-12: MDE Summer School[10] [78].

**Acknowledgments.** We would like to thank Antonio Cicchetti and Ludovico Iovino for the never ending discussions we had over the last few years about the topics covered by this paper (and mountaineering). Also, we are grateful to many colleagues, including Jean Bézivin, Jeff Gray, Richard Paige, Laurie Tratt, and Antonio Vallecillo, who shared their opinions and visions with us.

## References

1. Schmidt, D.: Guest Editor's Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (2006)
2. Bézivin, J.: On the Unification Power of Models. Jour. on Software and Systems Modeling (SoSyM) 4(2), 171–188 (2005)
3. Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 111–119. Carnegie Mellon University, Pittsburgh (2009)
4. Tratt, L.: Model transformations and tool integration. Jour. on Software and Systems Modeling (SoSyM) 4(2), 112–122 (2005)
5. Object Management Group (OMG): MOF 2.0 Query/Views/Transformations RFP, OMG document ad/02-04-10 (2002)
6. Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)
7. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a Modeling Language for Designing Web sites. Computer Networks 33(1-6), 137–157 (2000)
8. Di Ruscio, D., Muccini, H., Pierantonio, A.: A Data Modeling Approach to Web Application Synthesis. Int. Jour. of Web Engineering and Technology 1(3), 320–337 (2004)

[10] `http://www.sti.uniurb.it/events/sfm12mde/`

9. Cicchetti, A., Di Ruscio, D., Eramo, R., Maccarrone, F., Pierantonio, A.: beContent: A Model-Driven Platform for Designing and Maintaining Web Applications. In: Gaedke, M., Grossniklaus, M., Díaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 518–522. Springer, Heidelberg (2009)
10. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Model Differences for Supporting Model Co-evolution. In: Procs. MoDSE, 2nd Workshop on Model-Driven Software Evolution (2008)
11. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model Migration with Epsilon Flock. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 184–198. Springer, Heidelberg (2010)
12. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope - automating coupled evolution of metamodels and models, pp. 52–76 (2009)
13. Di Cosmo, R., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Supporting software evolution in component-based foss systems. Technical Report TRCS 003/2010, Computer Science Department, University of L'Aquila (2010)
14. Stevens, P.: A Landscape of Bidirectional Model Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
15. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting Parallel Updates with Bidirectional Model Transformations. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 213–228. Springer, Heidelberg (2009)
16. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc., San Diego (1985)
17. Favre, J.M.: Meta-Model and Model Co-evolution within the 3D Software Space. In: Procs. of the Int. Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICSM 2003, Amsterdam (September 2003)
18. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
19. Selic, B.: The Pragmatics of Model-driven Development. IEEE Software 20(5), 19–25 (2003)
20. Object Management Group (OMG): MDA Guide version 1.0.1, OMG Document: omg/2003-06-01 (2003)
21. Kent, S.: Model Driven Engineering. In: Butler, M.J., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
22. Favre, J.M.: Towards a Basic Theory to Model Model Driven Engineering. In: Procs. of the 3rd Int. Workshop in Software Model Engineering (WiSME 2004) (2004)
23. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Core Specification, OMG Document ptc/03-10-04 (2003), `http://www.omg.org/docs/ptc/03-10-04.pdf`
24. Object Management Group (OMG): Unified Modelling Language (UML) V1.4 (2001)
25. Object Management Group (OMG): XMI Specification, v1.2, OMG Document formal/02-01-01 (2002)
26. Seidewitz, E.: What Models Mean. IEEE Software 20(5), 26–32 (2003)
27. Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: Automated Software Engineering (ASE 2001), pp. 273–282. IEEE Computer Society, Los Alamitos (2001)
28. Mellor, S.J., Clark, A.N., Futagami, T.: Guest Editors' Introduction: Model-Driven Development. IEEE Software 20(5), 14–18 (2003)
29. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley (2003)
30. Aksit, M., Kurtev, I., Bézivin, J.: Technological Spaces: an Initial Appraisal. In: International Federated Conf. (DOA, ODBASE, CoopIS), Industrial Track, Los Angeles (2002)

31. Kleppe, A., Warmer, J.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
32. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems J. 45(3) (June 2006)
33. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., Varró-Gyapay, S.: Model Transformation by Graph Transformation: A Comparative Study. In: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica (October 2005)
34. OMG: MOF QVT Final Adopted Specification, OMG Adopted Specification ptc/05-11-01 (2005)
35. Xactium: Xmf-mosaic, `http://xactium.com`
36. Vojtisek, D., Jézéquel, J.M.: MTL and Umlaut NG: Engine and Framework for Model Transformation, `http://www.ercim.org/publication/Ercim_News/enw58/vojtisek.html`
37. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Metalanguages. In: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, pp. 264–278 (2005)
38. Didonet Del Fabro, M., Bezivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: A generic Model Weaver. In: Int. Conf. on Software Engineering Research and Practice (SERP 2005) (2005)
39. Cicchetti, A., Di Ruscio, D.: Decoupling Web Application Concerns through Weaving Operations. Science of Computer Programming 70(1), 62–86 (2008)
40. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 90–105. Springer, Heidelberg (2002)
41. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: A Bidirectional and Change Propagating Transformation Language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 183–202. Springer, Heidelberg (2011)
42. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Kowalski, R.A., Bowen, K. (eds.) Proceedings of the Fifth Int. Conf. on Logic Programming, pp. 1070–1080. The MIT Press, Cambridge (1988)
43. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
44. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
45. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
46. de Lara, J., Vangheluwe, H.: AToM$^3$: A Tool for Multi-formalism and Meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002), `http://link.springer.de/link/service/series/0558/bibs/2306/23060174.htm`
47. Varró, D., Varró, G., Pataricza, A.: Designing the automatic transformation of visual languages. Science of Computer Programming 44(2), 205–227 (2002)
48. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The Design of a Language for Model Transformations. Journal of Software and System Modeling (2005)
49. Konigs, A., Schurr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electronic Notes in Theoretical Computer Science 148, 113–150 (2006)

50. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a General Composition Semantics for Rule-Based Model Transformation. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS 2011. LNCS, vol. 6981, pp. 623–637. Springer, Heidelberg (2011)
51. Object Management Group (OMG): OCL 2.0 Specification, OMG Document formal/2006-05-01 (2006)
52. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, Technology, and Tools. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 483–505. Springer, Heidelberg (2003)
53. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon), `http://www.eclipse.org/gmt/epsilon`
54. Börger, E., Stärk, R.: Abstract State Machines - A Method for High-Level System Design and Analysis. Springer (2003)
55. Varró, D., Pataricza, A.: Generic and Meta-Transformations for Model Transformation Engineering. In: International Conference on the Unified Modeling Language, pp. 290–304 (2004)
56. Object Management Group (OMG): MOF 2.0 QVT Final Adoptet Specification v1.1, OMG Adopted Specification formal/2011-01-01 (2011)
57. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and Systems Modeling 8 (2009)
58. Steven Witkop: MDA users' requirements for QVT transformations, OMG document 05-02-04 (2005)
59. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective—GRACE Meeting Notes, State of the Art, and Outlook. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)
60. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 115–129. Springer, Heidelberg (2005)
61. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning (2004)
62. Tratt, L.: A change propagating model transformation language. Journal of Object Technology 7(3), 107–126 (2008)
63. Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)
64. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005)
65. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MoDELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
66. Cicchetti, A., Di Ruscio, D., Iovino, L., Pierantonio, A.: Managing the evolution of data-intensive web applications by model-driven techniques. Software and Systems Modeling (2011)
67. Gruschko, B., Kolovos, D., Paige, R.: Towards Synchronizing Models with Evolving Metamodels. In: Proceedings of the Workshop on Model-Driven Software Evolution, MODSE 2007 (2007)
68. Lin, Y., Zhang, J., Gray, J.: Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In: OOPSLA Workshop on Best Practices for Model-Driven Software Development (2004)

69. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology 6(9), 165–185 (2007)
70. Rivera, J., Vallecillo, A.: Representing and Operating with Model Differences. In: Objects, Components, Models and Patterns. LNBIP, vol. 11, pp. 141–160. Springer, Heidelberg (2008)
71. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. Upgrade, Special Issue on Model-Driven Software Development (April-May 2008)
72. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: ESEC-FSE 2007: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 295–304. ACM, New York (2007)
73. Cicchetti, A.: Difference Representation and Conflict Management in Model-Driven Engineering. PhD thesis, University of L'Aquila, Computer Science Dept. (2008)
74. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Dependent Changes in Coupled Evolution. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 35–51. Springer, Heidelberg (2009)
75. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Generic Model Transformations: *Write Once, Reuse Everywhere*. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 62–77. Springer, Heidelberg (2011)
76. Steel, J., Jézéquel, J.M.: On model typing. Software and System Modeling 6(4), 401–413 (2007)
77. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal Specification and Testing of Model Transformations. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 399–437. Springer, Heidelberg (2012)
78. Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.): SFM 2011. LNCS, vol. 7320. Springer, Heidelberg (2012)