Evolution of Domain-Specific Languages A Literary Study

Simon Van Mierlo

University of Antwerp, Belgium

Abstract. Model-Driven Engineering (MDE) is gaining popularity as an approach to building large software projects. Using models as first class entities in the development process, the level of abstraction is raised which leads to better understandability and maintainability of software artefacts. Current MDE solutions allow for the use of modelling languages to create models and transformations, either visually or textually. However, most approaches assume a static modelling language and do not support the evolution of such languages explicitely. This area of research has gained popularity in the last decade and in this paper current solutions to support the co-evolution of languages and their related artefacts in an MDE context are discussed. A set of criteria is introduced and used to classify these solutions. Using this classification, we try to dicover areas in which current solutions lack support. These findings are then used to point out areas in which future research is needed for a better support of language evolution.

Keywords: Model-Driven Engineering, Domain-Specific Languages, Evolution

1 Introduction

Throughout the years, model-driven engineering (MDE) [34] has grown in importance and has been adopted in several industrial software projects. The core concept of MDE is the creation of models using specialized modelling languages. Models raise the level of abstraction when compared to artefacts written in programming languages, leading to implementation-independent representations of different aspects of a software system such as requirements, design, logic, etc. Domain-specific modelling languages (DSMLs) are a subset of modelling languages which are used to represent the various facets of a software system using domain-specific concepts. The models created in a DSML can be used, amongst others, for automatic test case generation [7], code generation [12] and verification of properties [33].

Usually, a domain-specific language is created by a language developer through the construction of a metamodel [21]. The metamodel of a language defines all the concepts that exist in the language and which can be insantiated in models created using that language, as well as well-formedness rules that restrict the number of valid models. These well-formedness rules are also called the static semantics of the DSML [11]. Once the concepts and static semantics of a DSML are defined, models representing aspects of the system being built can be created using the DSML. These models are said to conform to the metamodel of the DSML. Besides models, another essential aspect of MDE is model transformation [24]. Model transformations are used to transform a model created in a certain language to another model, either in the same language (endogenous transformation) or in another language (exogenous transformation). These transformations can be used to attach semantics to a model, either by transforming it to a model in a language with known semantics (denotational semantics) or by a succession of transformations to models in the same language (operational semantics), which are capable of simulating or executing models [24].

As a DSML is tied closely to its domain and this domain or the requirements of the language users may change, the DSML has to be updated from time to time as well. This task is performed by the language developer by changing the metamodel of the language. This may, however, break the conformance relationship between the models and transformations created using the initial version of the language and the new version of the language. A concept may have been deleted in the metamodel, or a well-formedness rule altered. Once one or more conformance relations are broken between models and metamodel, the MDE system as a whole is said to be *inconsistent*. A primitive solution to re-establish consistency would be to migrate all models manually such that they conform to the new version of the metamodel. However, this is a tedious and error-prone job. Firstly, it may be possible that the models cannot be loaded into the normally used development environment anymore, because they do not conform to the newest version of the metamodel which has been installed in that environment. This results in the need to manually alter the source code of models and transformations, if this is at all possible. Secondly, correctness is not guaranteed. Even if all mod-

els conform to the new metamodel after migration, their semantics may have been changed involuntarily. Lastly, sometimes hundreds or thousands of models have been created in the initial version of the language. In that case, it may not be practically possible to migrate all of them.
A second solution to the problem of keeping an MDE system consistent, which is employed by programming language developers when such a language evoles, consists of ensuring the new version of the language is *backwards compatible* with the initial version and marking structures which should not be used anymore as *deprecated*. In doing so, no migration of models has to be performed. However, this is not a practical solution. It restricts the language developer in such a way that it may no longer be possible to keep the DSML as closely tied to the domain as possible. This is one of the main characteristics of a DSML and it should be possible to make such changes. An MDE system can be seen as a faithful representation of a domain and if that domain or its requirements change, the MDE system should evolve as a whole to represent that change.

There is thus need for a structured, semi-automatic method with which metamodels, models and transformations can be kept synchronized. This is called co-evolution of metamodels, models and transformations, since models and transformations have to evolve together with the metamodel. In the last decade, this topic has been given a lot of attention in the software engineering community, as it is seen as one of the obstacles to overcome in order for MDE to become a mainstream development method [25]. As such, different algorithms have been proposed, case studies have been performed and tools have been developed to investigate the possibility of supporting language evolution. These solutions differ greatly on several key aspects. For example, some approaches start from a difference model of the two versions of the metamodel and derive a migration transformation (semi-)automatically. Others provide dedicated evolution languages with which to specify the different steps taken to evolve the metamodel. In this paper, all approaches that have been proposed up to this point in time will be discussed. A categorization will be made using a set of criteria to compare these approaches. As such, we try to discover research topics that have not been suffi-

ciently given attention in the literature, which may lead to a better understanding and support for the co-evolution problem.

The general idea of model co-evolution is presented in Figure 1. A metamodel MM is adapted which results in a new version of the metamodel, called MM'. The adaptations are depicted as ΔMM. All conforming models M have to be migrated as to re-establish the conformance relationship with MM' which is achieved by executing transformation E.

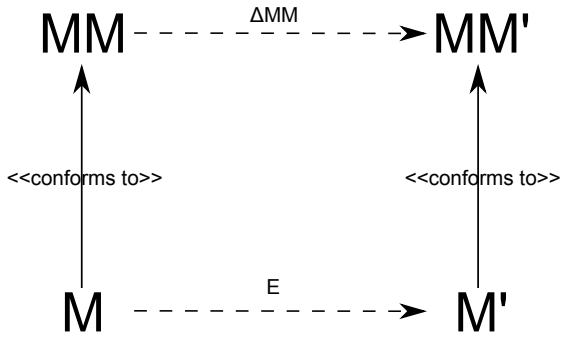

Fig. 1: Adaptation of a Metamodel

Figure 2 presents a more advanced case. There, a transformation T has been defined which maps models conforming to metamodel MM_D onto models conforming to metamodel MM_I. These metamodels are respectively called the domain metamodel and image metamodel of the transformation. In this example MM_D is adapted, which means conforming models M have to be migrated to reflect these changes, as was discussed previously. In this case, however, the transformation logic may need to be adapted as well. For example, if a new concept is introduced in the domain metamodel extra transformation logic will need to be added, and if a concept is removed from the domain metamodel certain transformation logic will have to be removed. The resulting transformation is called T' and is capable of transforming models conforming to the evolved metamodel MM' to models conforming to the image metamodel MM_I.

The paper is structured as follows. In Section 2, the different approaches are discussed. The solutions are classified using five eval-

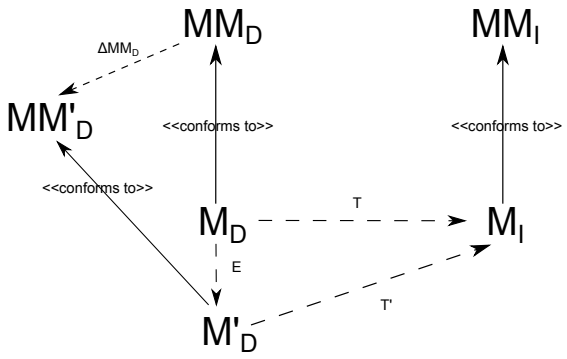

Fig. 2: Adaptation of a Domain Metamodel

uation criteria: the discussed types of metamodel adaptations and which of these are supported by the method proposed by the paper, the types of migration which are supported by the solution (model and/or transformation), the automatibility of the proposed solution and its completeness, whether intention (semantics/properties of a model) is preserved and whether tool support is present. Section 3 aggregates the findings of Section 2 and points out where current solutions lack functionality or which aspects of language evolution have not been discussed in the literature. Section 4 concludes the paper.

2 A Classification of Solutions

The solutions proposed for co-evolving metamodels, models and transformations differ significantly. In this section, we propose a set of criteria to categorize and differentiate between these solutions. This will allow us to decide where current solutions lack functionality or further validation of these solutions is needed and make suggestions for future work. The first criterium, which is discussed in Section 2.1, is how authors categorize the adaptations performed on a metamodel. If such a categorization is present, it is based on how the metamodel is adapted and what the consequences are for the related models and transformations. If a categorization is used in the paper and a solution is proposed, we will mention which of the adaptations the authors acknowledge are supported in their solution. Section 2.2 lists which migrations the proposed solutions support, which is either

model migration, transformation migration or both. In Section 2.3 the automatibility and completeness of each solution will be discussed. This is more often than not a trade-off: either the solution is complete but not fully automatic, or the solution is fully automatic but not complete. Section 2.4 discusses how intention (i.e. properties) of models are preserved during migration. Section 2.5, lastly, looks at tool support of the proposed solutions. This will prove to be one of the most important criteria of all, because a proposed solution cannot be properly tested without a tool that implements it.

2.1 Metamodel Adaptations

Classifying metamodel adaptations is done by most authors that propose solutions for the co-evolution problem. On the one hand, it is useful as a way to determine the effects of different types of adaptations on related artefacts like models and transformations, which leads to different types of actions to migrate these artefacts to the new version of the metamodel. On the other hand, it forces authors to acknowlegde which types of adaptations are supported by the proposed solution. If no classification is present, it is no trivial task to discover whether the solution supports all possible adaptations performed on a metamodel. In the next paragraph the classifications found in the literature are discussed. If a solution uses a certain type of classification, we will also discuss whether all types of adaptation are supported.

1. Non-Breaking and Breaking Changes: In [10] and [1], Gruschko et al. introduce a classification of metamodel adaptations into three categories: **Not Breaking Changes**, **Breaking and Resolvable Changes** and **Breaking and Unresolvable Changes**. The first category consists of adaptations which occur in metamodels but do not break the conformance relation between models and the metamodel. As such, models that conform to the initial version of the metamodel also conform to the new version of the metamodel. An example of this type of adaptation is the addition of a non-mandatory class in the metamodel, as all models not containing an instance of this class conform to the new version of the metamodel as well. The second category consists of adaptations which breaks the conformance relation between the models conforming to the initial version

of the metamodel and the new version of the metamodel, but can be resolved by automatic means. An example adaptation is the renaming of a class. This change results in all models containing an instance of the renamed class to not conform to the new version of the metamodel anymore. It is however possible to create an automatic renaming mechanism that renames all instances of the renamed class to reflect the name change. The last category consists of adaptations that break the conformance relation between models and metamodel and are not resolvable automatically. An example change is the addition of a mandatory class to the metamodel. None of the models conforming to the initial version of the metamodel will contain an instance of this class, which means an instance of the class will need to be added to every model. However, we cannot automatically decide how and where to add an instance of a class and as such a developer will have to manually edit each model.

This change classification has become a popular choice for authors to use and is the basis for many of the proposed solutions. In the paper by Gruschko et al. a solution is proposed which uses the change classification to take different types of actions based on the type of metamodel adaptation. It consists of a change recording phase in which a difference model is built, a classification phase in which the adaptations represented in the change model are classified in the categories presented here and a manual edit phase for unresolvable changes. After that a transformation is constructed which migrates instance models. The solution proposed by the paper thus supports all adaptations using the categorization the authors have introduced.

In [3], Cicchetti et al. use a similar approach. There, a difference model is constructed as well and the adaptations are classified. However, this classification leads to two non-overlapping difference models which are used to construct two migrations (one automatically, the other with the intervention of the user) which are executed concurrently to migrate models. This technique can be applied if all adaptations in the two difference models are independent of each other. If a dependency exists, certain migration steps need to be executed before others. The authors resolve this problem in [4], which makes this solution complete in the sense that it supports all categories of changes.

The classification has also been used for solutions that migrate trans-

formations. In [9], Garcia and Díaz use the classification to support transformation migration. A difference model is created to represent the metamodel adaptations, which is used by an ATL higher-order transformation to migrate the transformation. All categories of changes are discussed and supported by their solution.
In [26], Meyers and Vangheluwe build a framework that supports the evolution of modelling languages in general. This encompasses metamodels, models, transformations and their visual representations. They make use of the classification proposed by Gruschko and support all of the categories of changes. They achieve this by breaking down the evolution of languages into primitive scenarios, which encompass the consequences of evolution and the required remedial actions. By combining these primitives in a high level framework, all possible evolutions are supported.
Van Den Brand et al. extend the original classification in [37] by dividing **Breaking and Unresolvable Changes** into **Breaking and Semi-Resolvable Changes** and **Breaking and Human-Resolvable Changes**. The first category encompasses metamodel adaptations that cannot be resolved automatically but can be resolved by configuring the evolution process. The adaptations in the second category can only be resolved by a user in a differences-resolution environment. All of these changes are supported by the solution proposed in the paper.

2. Metamodel and Model Independent and Specific Changes: In [14], Hermannsdoerfer et al. introduce COPE, a tool used to incrementally evolve a domain-specific language by employing coupled operators. They use a change classification which is similar to the one described in the previous paragraph. Metamodel adaptations are classified into four categories: **Metamodel Only Changes**, **Metamodel Independent Changes**, **Metamodel Specific Changes** and **Model Specific Changes**. The first category encompasses changes that only have an effect on metamodels. This category corresponds to **Not Breaking Changes** discussed above which means no co-evolution has to be performed on models. For the second category, generic and reusable coupled operators can be defined. This is a core concept in COPE: the more reusable coupled operators that are defined, the more useful the solution becomes, because it saves

the user from having to define his own specific operators. An example of such an operator is the renaming of a property of a class in the metamodel. It is possible to define a generic, reusable coupled operator for this adaptation as the operations performed on the metamodel and conforming models are always the same, irrespective of the specific metamodel the operator is applied on. In [16], an extensive catalog of coupled reusable operators is defined to demonstrate the usability of the operator-based approach. The third category encompasses metamodel adaptations which need the definition of a custom coupled operation because the operator cannot be reused for other metamodels. They can, however, be defined in a model-independent way. COPE allows the user to define metamodel-specific coupled operators by applying a set of primitives for both metamodel adpatation and model migration. This set of primitives is complete which means every possible metamodel adaptation and model migration can be specified with them. The last category consists of metamodel adaptations that have to be resolved on a per-model basis. This means no coupled operator can be defined (either reusable or custom) that evolves the metamodel and migrates the instance models automatically. The developer has to manually intervene in the migration process, which was not supported in the first version of COPE. In [15], Herrmannsdoerfer et al. introduce constructs that allow the user to choose for each model which action has to be taken to migrate it. As such, COPE supports all types of changes described by this classification.

3. Metamodel Isomorphism, Extension, Projection, Factoring: In [17], Hössler et al. classify metamodel adaptations based on relations between the two versions of the metamodel. The relations discussed in the paper are **Metamodel Isomorphism**, **Metamodel Extension**, **Metamodel Projection** and **Metamodel Factoring**, which are defined as follows. Two metamodels are said to be **isomorphic** if the set of their instance models is equivalent. All instances conforming to the initial version of the metamodel also conform to an **extension** of that metamodel. This resulting metamodel is then called a **super-metamodel** of the original metamodel. Conversely, the original metamodel is a **sub-metamodel** of the resulting metamodel. A metamodel is a **projection** of another metamodel if the

size of the set of instance models reduces by deleting a metamodel element. **Metamodel Factoring** encompasses more complex patterns of metamodel evolution, including property movement and amalgamation and class splitting and amalgamation. The paper mentions how to deal with each of the relations, but no proof is presented that every possible metamodel adaptation can be categorized using this classification. Because of this, we cannot conclude that the solution is complete.

4. Construction, Destruction and Refactoring In [39], Wachsmuth proposes one of the most elaborate classifications of metamodel adaptations. First, preservation properties for metamodels are presented by defining metamodel relations like equivalence, variation, sub- and super-metamodel. These are then used to derive semantics- and instance-preservation properties for metamodel relations. Then, a set of edit operations are classified into **Construction**, **Destruction** and **Refactoring** operations. Each operation is associated with a semantics-preservation property which defines what the result on the metamodel and its instance models is when the operation is executed. For each of these categories, co-evolution operations are defined which migrate models in response to the changes performed on the metamodel. In that sense, all adaptations which are described are supported by the solution. However, as will be seen further on, it may be possible that not all possible metamodel adaptations can be categorized using this classification. Performing an additive change may not necessarily mean that the set of instances becomes larger. For instance, if a relation is added between two classes with multiplicity 1..1, all instance models containing instances of those classes will have to be adapted, adding a relation between the instances. This means the set of valid instances reduces instead of increases. None of the metamodel relations defined by Wachsmuth describe this.

5. Fully Automated, Partially Automated and Fully Semantic In [22] a solution is proposed for the semi-automatic migration of model transformations. There, a classification is made which consists of three categories: **Fully Automated Changes**, **Partially Automated Changes** and **Fully Semantic Changes**. The first category

consists of adaptations for which migrations can be automatically constructed, like the renaming of an attribute. The second category consists of adaptations for which information is lacking to migrate transformations. The example of deleting an attribute is given. If the attribute is used in a transformation, we do not know how the transformation should be adapted to compensate for the missing attribute. It is the task of the developer to fill in this missing semantic information. The last category consists of adaptations for which the developer needs to supply all semantic information. An example for such an adaptation is the addition of an element. No transformation rules are yet defined for this new element and as such the developer will have to supply all needed information in the transformations. The proposed solution performs an automated pass for adapations in the first category, followed by a manual pass for adapations in the last two categories.

In Table 1 a set of commonly occuring metamodel adaptations are classified according to the five aforementioned classifications. It attempts to clarify the presented classifications and by no means forms a complete set of metamodel adaptations. As can be seen, some classifications cannot deal with certain metamodel adaptations. These cells are marked with a question mark.

2.2 Supported Migrations

When applying an MDE development process and using its full potential, a number of models and transformations are created. As previously stated, co-evolution for both models and transformations should be supported in order for MDE to become an accepted development method. In this section, the proposed solutions will be categorized according to the types of migration they support. This is either model only, transformation only or both.

Model Only: In [36] a domain-specific visual language is presented which supports the evolution of domain-specific modelling languages. This language is used to define patterns which describe the migration steps to perform when migrating models from one version of a

Adaptation	**1**	**2**	**3**	**4**	**5**
Generalize Metaproperty	Not Breaking	Metamodel-Only	Super-Metamodel	Construction	Fully Automated
Add (non-obligatory) Metaclass	Not Breaking	Metamodel-Only	Super-Metamodel	Construction	Fully Automated
Add (non-obligatory) Metaproperty	Not Breaking	Metamodel-Only	Super-Metamodel	Construction	Fully Automated
Extract (abstract) Superclass	Not Breaking	Metamodel-Only	MM Isomorphism	Refactoring	Fully Automated
Eliminate Metaclass	Breaking and Resolvable	Metamodel-Independent	MM Projection	Destruction	Partially Automated
Eliminate Metaproperty	Breaking and Resolvable	Metamodel-Independent	MM Projection	Destruction	Partially Automated
Push Metaproperty	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Destruction	Partially Automated
Flatten Hierarchy	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Destruction	Partially Automated
Rename Metaelement	Breaking and Resolvable	Metamodel-Independent	MM Isomorphism	Refactoring	Fully Automated
Move Metaproperty	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Refactoring	Partially Automated
Extract Metaclass	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Refactoring	Fully Semantic
Inline Metaclass	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Refactoring	Partially Automated
Add Obligatory Metaclass	Breaking and Unresolvable	Model-Specific	?	?	Fully Semantic
Add Obligatory Metaproperty	Breaking and Unresolvable	Model-Specific	?	?	Fully Semantic
Pull Metaproperty	Breaking and Unresolvable	Model-Specific	MM Factoring	Construction	Fully Semantic
Restrict Metaproperty	Breaking and Unresolvable	Model-Specific	Sub-Metamodel	Destruction	Fully Semantic
Extract (non-abstract) Superclass	Not Breaking	Metamodel-Only	MM Factoring	Construction	Fully Automated

Table 1: Adaptation Classifications

metamodel to the next.
A similar method was used in the construction of the tool *Lever* [28]. Language evolutions in *Lever* are textually specified and used for both metamodel evolution and model migration. Model migrations created by *Lever* transform the abstract syntax graph representation of a model created in any version of the language to the last version of the language by keeping track of the history of the language and the evolution operations performed on the metamodel of the language.
In [38] a domain-specific transformation language (DSTL) is derived from the metametamodel the metamodel conforms to automatically. This language allows to specify evolution scenarios for the language. To support co-evolution, a user-defined mapping between primitive evolution operators in the DSTL and corresponding model migration steps has to be provided. From the evolution scenario and user-defined mapping a model migration tranformation is constructed automatically.
In [27] the model change language (MCL) is used to define metamodel evolution and model migration patterns. MCL is a visual language which is used to specify the patterns which define the evolution scenario. These patterns contain information to evolve the metamodel and co-evolve conforming models.
Epsilon Flock [30] is a pattern language which is used to specify patterns that define model migration scenarios. The language developer creates a model migration transformation using Flock which is capable of migrating models from one version of the metamodel to the next.
In [23] existing in-place transformation languages are used to define model migrations. The first step in the process is to merge the two versions of the metamodel, as both elements from the initial version and the evolved version will be used in the transformation patterns. Using this merged metamodel, a user can define the necessary rules for instantiating metamodel elements introduced in the evolved version of the metamodel. The last step in the process is to remove elements that are no longer present in the evolved version of the metamodel automatically.
In [10] and [1] model migration is approached differently. In these papers, a process model is created with which a metamodel and its

instance models are co-evolved (semi-)automatically. First, a change model is computed which represents the metamodel adaptations. These adaptations are classified after which user input is gathered if needed. Then, the necessary model migration algorithms are determined and the transformation is executed.
In [3] and [4] a similar method is employed. The main difference is that after change classification, the difference model is split into two non-overlapping difference models based on the automatibility of the migration derivation. Two migration transformations are derived from these change models and executed in parallel to migrate models.
In [8] metamodel adaptations are detected by comparing the two versions of the metamodel and employing a set of user-chosen and/or user-defined heuristics which has to be configured for each evolution scenario. The resulting difference model is then mapped onto a migration transformation.
In [39], metamodels are evolved by stepwise adaptation. This is done by so-called coupled operators, which consist of the metamodel adaptation and corresponding model migration transformation. The same approach is used in COPE [14, 15], where reusable coupled operators are defined in a library. A facility to create custom coupled operators is also provided.

Transformation Only: There are only a few solutions providing only transformation migration. In [9] a semi-automatic method is developed which is based on a generated difference model. The metamodel adaptations are classified, after which transformations are generated automatically or with the help from the user if semantic information is missing.
In [22] the MCL is used to specify metamodel adaptations. Then, transformation migrations are derived automatically where possible or provided by the user in the form of MCL patterns.

Model and Transformation: In [26] both model and transformation migration are discussed. As previously mentioned, the authors create a framework for the evolution of languages which is as complete as possible. To evolve models and transformations a pipeline is built which defines the different steps of the evolution process. These steps

can be found in other solutions as well: change detection and representation, automatic transformation generation together with user input gathering for unresolvable changes and transformation execution.
[40] discusses both model and interpeter migration. The authors present an approach in which a specification of the metamodel adaptation is mapped onto model and interpeter migration transformations.

2.3 Automatibility and Completeness

Automatibility of a solution pertains to how much user intervention is needed: either the solution is fully automatic, only based on user input or a combination of the two. Completeness is a characteristic of a solution which is closely tied with automatibility. We regard a solution as complete if all possible evolution scenarios are supported. It is closely tied with automatibility because a solution which is fully automatic often is not complete and a solution which is only based on user input often is complete, but requires a lot more effort from the user.

Fully Automatic: Fully automatic solutions usually start from a model that captures the adapations performed on the metamodel. This model can either be the result of a change recording mechanism that keeps track of all adaptations perfomed on the metamodel or an algorithm that compares the two versions of the metamodel after the changes have been performed. This change model is then used to derive a migration transformation which is subsequently executed on models to migrate them to the new version of the metamodel.
In [8], Garcés et al. attempt to fully automatically migrate instance models. There, two versions of the metamodel are compared by an algorithm which is the composition of a set of heuristics. These heuristics have to be chosen by the language developer on a per-metamodel basis to get the best resulting difference model. It is however a nontrivial task to decide a priori which of the heuristics are required to produce the best results. In the paper, the derivation of the adaptation transformation is given little attention and it is assumed this mechanism works correctly. These arguments lead to the conclusion

that the solution cannot be deemed complete.
In [40], a generative approach to intepreter evolution is proposed. In their solution, the authors map a specification of the metamodel adaptations onto interpreter and model migration transformations. They require that a formal specification of metamodel changes has to be present, either provided by the user or computed by a differencing technique from the two metamodel versions. The authors state that more research has to be performed to evaluate the feasibility of the approach. It is thus not complete.

Only User Input: This category consists of solutions that require the user to define a model migration transformation manually. This migration is either specified in an existing transformation language or in a newly created domain-specific language for the domain of model migration.
In [36] a domain-specific visual language is created to specify visually how models should be migrated in response to metamodel adaptations. A sequence of transformations is created by the user and concatenated in a user-specified order. The models are then migrated by performing each transformation in this order. Primitives for the creation, deletion and mapping of model elements are provided in the language. It is not clear, however, whether these primitives form a complete set and can be used to specify arbitrary complex migration scenarios. In the example given a rather trivial co-evolution problem is solved and there is, for example, no change present that could be described as 'breaking, not resolvable' which requires user input for each model that is to be migrated. As such, we can conclude this solution is not complete.
In [28], the tool *Lever* is created. In *Lever*, model migrations are defined textually by the user. The tool keeps track of the history of the metamodel, which allows a model created in every version of the language to be loaded into the tool by first interpreting it using the version of the language it was created in and then migrating it using the user-defined migration transformations. This solution, however, does not include mechanisms to deal with model-specific changes, i.e. those changes for which user intervention is required on a per-model basis. As no categorization of metamodel adaptations is employed in the paper, it is a challenging task to prove the solution

can deal with every possible metamodel adaptation. Therefore, we can conclude that this solution is not complete.
In [27], the model change language (MCL) is introduced. MCL is used to define patterns for model migration using both versions of the adapted metamodel. A pattern in MCL consists of a left hand side, defining which element of the initial version of the metamodel should be matched and a right hand side, where modified elements in the new version of the language can be used. The user defines relations between these elements and a model migration is subsequently derived. The paper is not extensive and it is not clear whether all complex migration scenarios are supported by MCL. We therefore conclude that this solution is not complete.
In [31] a different approach altogether is investigated. The authors observe that models conforming to the old version of an evolved metamodel often cannot be loaded into the normal development environment used to create and edit models after installing the new version of the metamodel. To counter this, the authors propose a solution where models are bound to a generic metamodel. This makes it possible to load all models, even non-conforming ones. When an inconsistency is detected in a model, the user is notified and markers are placed in the places which cause the inconsistency. The user can then edit the model which has been transformed to the Human Understandable Textual Notation (HUTN). The solution is complete but may require a lot of effort from the user as each model has to be migrated manually.
Epsilon Flock, introduced in [30] is a model-to-model transformation language which can be used to define migration patterns. The authors analyze ATL, COPE and Ecore2Ecore to derive requirements for their tool, which tries to combine the advantages of each tool and avoid their disadvantages. The result is a pattern language which consists of two elements that either migrate types or delete them. It automatically copies elements conforming to the new version of the metamodel language, which is called a conservative copy strategy in the paper. Flock is complete as it uses the Epsilon Object Language, which is expressive enough to perform any evolution scenario.

Combination: Solutions that are a combination of the previous two categories usually use the same approach as can be found in the

fully automatic solutions, but consist of an automatic phase and a phase in which the user can provide missing semantic information. These approaches are logical consequences from the act of classifying metamodel adapations based on their effect on conforming models. As we have seen in Section 2.1, authors always make the distinction between adapations for which the remedial action can be deducted automatically and those for which user intervention is required. If one can then accurately detect all adaptations performed on the metamodel and classify those adaptations, it is possible to divide the generation of the migration transformation into two non-overlapping parts.

A second category consists of operator-based solutions, whereby metamodels are adapted by the successive application of coupled operators. Some of these operators can be reused and information regarding model migration is attached to these reusable operations, permitting the automatic deduction of a migration transformation. Other operators are so specific to a certain language evolution scenario that they have to be defined by the user. That is why operator-based solutions are classified as solutions which are a combination of automatic solutions and solutions based on user input: a user evolves the metamodel by applying operators which are either reusable or user-defined, and a migration transformation is (semi-)automatically deduced.

In [10] and [1] a process model is constructed which performs the steps of the first approach discussed in this paragraph. First, a change model is constructed, either by change recording during the editting of the metamodel or by direct comparison of the two versions of the metamodel. Then the changes are categorized and transformations are constructed, either automatically for resolvable changes or by user input gathering for unresolvable changes. Then the transformation is executed to migrate the model. This solution is complete in the sense that it should be able to handle all types of changes, yet it is incomplete because the change detection algorithms and the algorithms to automatically derive transformations from metamodel adaptations are largely omitted.

In [3] a similar approach is used. However, the classification of changes in the proposed approach is made explicit by dividing the change model into two non-overlapping change models, one for the auto-

matically resolvable changes and one for the non-resolvable changes. From these models, two transformations are generated which can be run concurrently to migrate models to the new version of the metamodel. A challenge recognized by the authors with this approach is dealing with dependent changes. They therefore propose an algorithm in [4] to resolve these dependencies by scheduling transformation executions correctly. This approach is complete in the sense that all possible scenarios should be supported. However, the authors also mention the need for validation on a large set of metamodels and models, which has yet to be performed.

In [39] the concept of coupled operations is first explored. These operations are first described on the metamodel level as QVT Relations that define the adaptations performed. These can be used for the stepwise adaptation of metamodels. Next to these adaptations, model co-adaptations are defined in the form of parameterized QVT Relations. A metamodel adaptation transformation can then call one of these co-adaptation transformations with the correct parameters to create a model migration transformation. It is possible for users to intervene in this process by defining new adaptation transformations, co-adaptation transformations or OCL queries to fill in missing semantic information. It is therefore a complete process in the sense that any scenario *could* be supported, although only a handful of adaptation transformations are presented as examples in the paper.

COPE is a tool that supports the co-evolution of metamodels and models using coupled operators. It is introduced in [14] and expanded in [15]. In [16] an extensive catalog of reusable coupled operators is presented with which a language developer should be able to specify most common language evolution scenarios. A coupled operator is an operator which performs the metamodel adaptation and contains the migration transformation steps which need to be carried out in order for models to be migrated to the adapted metamodel. In COPE, a series of adaptations is performed by executing coupled operators on a metamodel. During this process, the corresponding model migrations are recorded. Once a developer is done editing the metamodel, these model migration steps are combined into one migration transformation which can be executed to migrate models to the new version of the metamodel. As COPE supports the definition of custom coupled operators and is capable of asking for user input

when additional semantic information is needed during the migration process, this is a complete solution.
Vermolen et al. employ the use of a domain-specific language for evolving metamodels in [38]. The DSL is created starting from the metametamodel which the metamodel conforms to. This languages contains primitives for adding, removing and modifying metamodel elements. Next to the DSL, a user-defined mapping has to be supplied that maps metamodel adaptations to the required model migrations. Using these elements, a metamodel adaptation is created in the DSL by the language developer, after which a model migration transformation is automatically calculated. The solution is not complete, as the definition of the user-supplied mapping is not given any attention and as such it is not clear whether every possible mapping can be specified. There is no mention of support for user input during the migration process either (for model-specific adaptations).
In [9] a solution is provided which evolves transformations semi-automatically in response to a metamodel adapation. Only adaptations of the source metamodel are considered, as such it is not complete. The approach classifies changes and either automatically generates the full transformation code or only a skeleton which requires the user to fill in the missing semantics.
Another approach to semi-automatically migrate transformations is described in [22]. There, the MCL is used to describe the adaptations performed on the metamodel. Subsequently, migration transformations are semi-automatically deduced. A classification is presented which divides adaptations into categories based on their automatibility. Some of the adaptations, like the renaming of a metamodel element, can be resolved automatically. For others, only the skeleton in the resulting transformation can be generated. For others, the user needs to specify the complete transformation from scratch. The solution is not complete as the authors do not cover this last class of adaptations, which are called fully semantic changes in the paper.
In [23] the use of existing in-place transformation languages for specifying model migration transformations is explored. The user has to specify rules for elements that have been changed in the new version of the metamodel. For this purpose, the two versions of the metamodel are merged as to allow the use of both outdated and new elements of the metamodel in these rules. After executing the

user-defined rules, outdated elements are removed from models automatically. There is no support for automatic generation of rules for resolvable changes and the authors mention this as future work. However, the solution is complete as it should be possible for users to specify every pattern needed to perform migration of models. The approach is not formally validated in the paper.
In [26] the authors attempt to provide a complete framework for the evolution of modelling languages. The framework supports the semi-automatic migration of models and transformations in response to metamodel adaptations. In the paper, every possible evolution scenario is exhaustively explored. The solution uses a migration pipeline which consists of all necessary steps to be performed in order to migrate all artefacts related to an evolved metamodel. First, an intermediate metamodel is created which is the result of merging the two versions of the metamodel. Next, from the automatically resolvable adaptations, migration transformation steps are constructed. If needed, user input is gathered and the migration transformation is executed. It is complete, as it is able to handle every possible evolution scenario using the proposed approach.
In [37] a generic metametamodel is constructed. All metamodels can be transformed in such a way that they conform to this metametamodel. After this transformation, metamodels can be regarded as a special kind of model. This allows the use of existing model differencing techniques to obtain a difference model by comparing the two versions of the metamodel. Using this difference model, the authors generate a difference model for each model that needs to be migrated. This difference model captures the adaptions that have to be performed on the model in order for the conformance relation to be re-established. This approach is complete, as it can deal with all possible types of changes and is extensible by providing user-defined transformations. A large validation study is present which shows the solution is applicable in real-life language evolution scenarios.

2.4 Intention Preservation

Intention preservation is a somewhat overloaded concept. In [26], the authors introduce the concept of *continuity* of software language evolution. This means that the system is consistent (models conform

to their metamodel after metamodel adaptation) and semantically equal to its previous version, modulo intented changes. *Semantics* in this context mean the properties a certain model has and not, as was previously defined, the denotational or operational semantics attached to a modelling language by model transformations. The properties of a model correspond to the properties the model has in its semantic domain. For instance, a possible property for a model created using a modelling language which is mapped onto Petri Nets is *liveness*, meaning it never reaches a deadlock state. In the paper it is mentioned that in order to support continuity, there has to be some mechanism in place in order for the properties to be checked. As an example, a constraint on a toy language for constructing train networks is given: if a constraint exists on the initial version of the language that no two trains should ever be on the same rail, this should be the case in every subsequent version of the language. This could be checked by transforming the models to a Petri Net model and running a reachability analysis. However, this is a rather ad-hoc way of checking properties and cannot serve as a general mechanism to ensure continuity. As properties are checked after migration is performed, it is not possible to decide whether a migration transformation will preserve the properties of a model before it is executed. The set of models on which properties are checked is furthermore only a subset of the set of all models which can be created with the modelling language, and it is not possible using this method to formally prove that a transformation preserves properties, before or after migration has been performed. More research has to be done in this area to discover possible solutions to overcome the challenge of continuity.

Other methods exist to ensure semantic equivalence of models before and after migration. However, they often defer the responsibility to someone or something else. In the case of manual specification methods (see Section 2.3) the user is responsible for providing the correct migration patterns and ensuring semantic equivalence. In [36] this fact is mentioned and the only solution provided is to spend a lot of effort on the development of the migration transformation. There is, again, no way to check a priori whether a migration will preserve certain semantic properties of a model.

When using (semi-)automatic methods of generating a transforma-

tion migration, the correct working of the algorithm depends largely on the change recording or differencing technique which is used to build the difference model. If all changes are represented faithfully, which requires the difference metamodel to be complete and the technique to build the difference model to be correct, it should be possible to define remedial actions for every possible scenario and ensure correctness with respect to semantic preservation. However, lots of differencing techniques have issues with quite trivial edit operations. Let's say we want to move a property from a class to its superclass. We therefore delete the property of the class and create a new one in the superclass with the same name and data type. To ensure semantic equivalence, we would like that the value of each instance of the property be retained in the migrated models. However, the difference model might not be able to differentiate between this edit operation and two unrelated edit operations: removing a property and creating a property. If it detects these last two, there is no way for the system to know that it should keep the value of the properties and semantic information will be lost. In [8] a substantial effort has been made to accurately detect metamodel adaptations. However, the process presented in that paper has to be configured on a per-case basis. The process uses a set of heuristics to compute the difference model. One of the main characteristics of a heuristic is that it can approach perfection but never attain it. This may be unacceptable in certain applications that require rigurous checking of properties using a formal method.
We can conclude that semantic preservation of models is currently not supported by language evolution solutions and needs further researching.

2.5 Tool Support

Tool support is one of the deciding factors whether a proposed technique will be succesful and a way to check its functionality. In this section, we will look at which solutions are implemented in tools, which we divide into prototypes and fully functional tools.

Prototypes: As was discussed earlier, in [28] the tool *Lever* is constructed. This is a prototypical implementation of the techniques

presented in the paper which tries to automatically maintain a layered DSL which undergoes adaptations. The authors have performed a small case study using the tool, but it is not extensive enough to conclude that this tool is mature and ready to be used on industrial-sized projects.
In [10] a prototype is built which implements the three-step process of change recording and exporting, classification and transformation generation and transformation execution. The prototype only supports the renaming of structural features.
In [38] a prototype is constructed which implements the architecture presented in the paper and has been applied to the domain of data modelling. The solution in the paper automatically derives a DSL in which evolution patterns for metamodels can be defined and derives model migration transformations from these patterns and a user-defined mapping which specifies how metamodel adaptations should be mapped onto model migrations.
Cicchetti et al. have implemented their approach, which is described in the previous sections, in a prototype tool [3] [4] on the AMMA [2] platform.
The AMMA platform was also used by Garcés et al. to create a prototype for their solution in [8]. They use the AtlanMod Model Weaver (AMW) [5] to work with matching models and ATL is used to implement the heuristics and the higher-order transformation which is responsible for generating the migration transformation.
In [37] a prototype tool is built which is tested on 10 metamodels, each having 10 conforming models. However, the amount of tested metamodel adaptations is rather small.

Fully Functional: A fully functional tool based on the use of coupled operators is COPE [14, 15]. It is integrated into the EMF metamodel editor and provides all the necessary functionality for metamodel and model co-evolution. Users can edit a metamodel using a context-aware editor which presents the library of reusable coupled operators. Custom operations can be created, edited and subsequently applied on metamodels. The tool keeps track of the history of the metamodel adaptations. Once a user is done editting the metamodel, he releases a new version of the language which automatically creates a migrator capable of migrating models to the new version of the language.

COPE has been tested thoroughly and appears in a couple of industrially sized case studies [13, 29].
In [31], a tool is built which binds models conforming to an EMF metamodel to a generic metamodel. The authors then generate a report which states whether the model is inconsistent with its metamodel and if so, which parts of the model are responsible. The user can edit the inconsistent models as to re-establish the conformance relationship.
Evolving transformations can be accomplished by the tool built in [22]. The tool is implemented in the GME/GReAT toolset and has been tested in an industrial environment. Future work which is mentioned in the paper consists of providing tool support for the addition of missing semantic information. In the current version of the tool, this information has to be added manually, which may mean a lot of work.
Lastly, Epsilon Flock [30] is a model-to-model transformation language built atop of the Epsilon Object Language (EOL) [19], which is the core of the Epsilon platform [20]. Flock is used for defining model migration patterns and has proved its worth in industrial sized case studies [29].

3 Analysis

In this section the results of the previous section are analyzed. We will attempt to discover open research questions which should lead to a better support for the co-evolution of metamodels and their related artefacts when discussed in future work. In each of the following subsections, four pairs of evolution criteria will be the basis for four matrices. The matrices consist of an x-axis and a y-axis, each one used for a particular criteria. For each criteria, a scale has been introduced which rates the solution based on the findings in the previous section. The criteria used as axes are *Migration Support* (Model, Transformation, Both), *Tool Support* (Prototype, Implemented, Tested), *Automation* (Only User Input, Fully Automatic, Combination), *Algorithm* (Manual, Operator Based, Differencing, Change Recording) and *Example* (Toy, Full Fletched). In Section 3.1, migration support and tool support of solutions are combined into a matrix. In Section 3.2 the same is done for automation and tool support. In Section

3.3, the matrix axes are automation and tool support. Using these three matrices, we can conclude which proposed solutions lack tool support. Lastly, in 3.4, we take a look at the types of supported migrations of each solution and whether an example is present.

3.1 Migration Support vs. Tool Support

A solution can either support model migration, transformation migration or both. In this section, a matrix is created (see Figure 3) which lists the solutions discussed in Section 2 along the x-axis according to this criterium and along the y-axis according to whether tool support is present. If tool support is present, three categories of tools are distinguished: a prototype is a tool which implements the solution but has only been used to migrate toy examples or examples from which cannot be derived that the tool can deal with arbitrarily complex metamodel adaptations. An implemented tool is a tool which implements the solution faithfully and should be able to handle most or all evolution scenarios. A tested tool is a tool which is implemented and tested on a large, industrial-sized project.

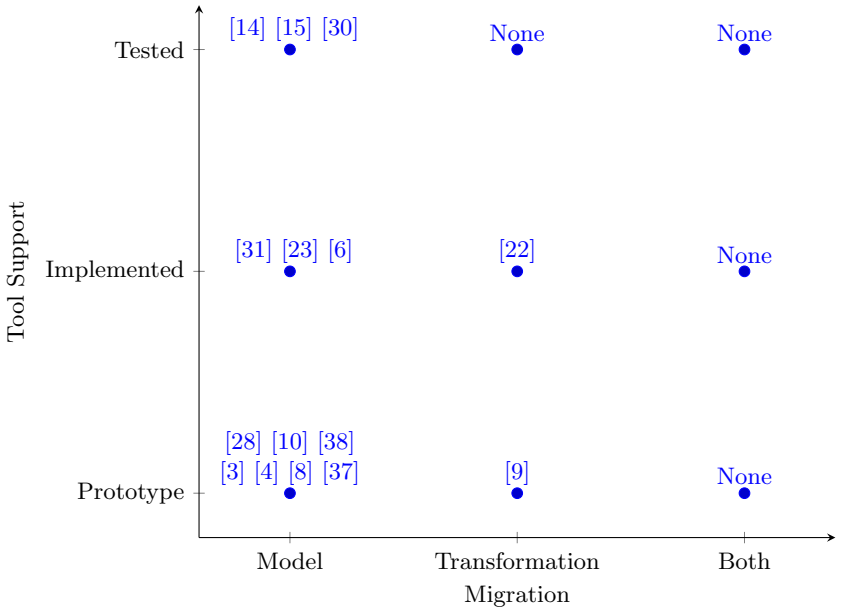

Fig. 3: Migration Support vs. Tool Support

As we can deduct from Figure 3 major progress has been made developing tools that support model migration. This is to be expected as most of the literature on metamodel evolution deals with the migration of models only. Implementations that have been tested thoroughly are COPE and Epsilon Flock. In [29] these two tools apper in a study which compares four tools capable of co-evolving metamodels and models. The tools are used on two co-evolution scenarios: one toy example (a Petri Net metamodel) and a larger example taken from a real-world model-driven development project. Transformation migration is an area which has not been explored extensively. In [9] a semi-automatic method is proposed, generating ATL transformations from a difference model resulting from the comparison of the two metamodel versions. [22] uses the MCL to describe metamodel adaptations and corresponding transformation migrations. Transformation migrations are constructed semi-automatically in two phases: an automatic phase and a phase in which the user can supply missing semantic information. In [26] a third approach to transformation migration is explored. The authors use function composition to create the evolved transformation from the original transformation and the migration transformation generated to migrate models conforming to the initial version of the metamodel. For example, in the case of image evolution (refer to Figure 2 for a case of domain evolution), the resulting transformation T' = $E \circ T$. However, because of the *projection problem* T' = $E \circ T$ does not always hold for image evolution. The projection problem states that E does not necessarily map models of the old version of the language to the complete set of models in the new version of the language. For instance, if a non-obligatory concept is introduced in the language, E will not migrate models to models containing instances of this class. As such, $E \circ T$ will not explore the full power of the new version of the image language, which may be undesirable. User input is required in these cases to fill in missing semantic information. In the paper, however, only a conceptual framework is built and therefore it is not listed in the matrix presented in this section. This section can be concluded by stating that no verified implementation of transformation migration has been presented yet. No attempt has been made to create a tool which supports both model and transformation migration.

3.2 Automation vs. Tool Support

In this section, we will discuss tools and which level of automation they support. The automation levels of algorithms we distinguish are user input only, fully automatic and a combination of these two, which are semi-automatic solutions. These levels correspond to the ones discussed in Section 2.3. Unlike some of the criteria presented in this paper, a solution is not inherently better if it uses a certain level of automation instead of another one. As mentioned in Section 2.3, each level of automation brings its challenges and advantages: a fully automatic solution requires no user input, but it is challenging to create a complete solution using this technique. A solution which requires the user to manually specify the migration steps requires a lot of involvement of the user, but it is easier to achieve completeness.

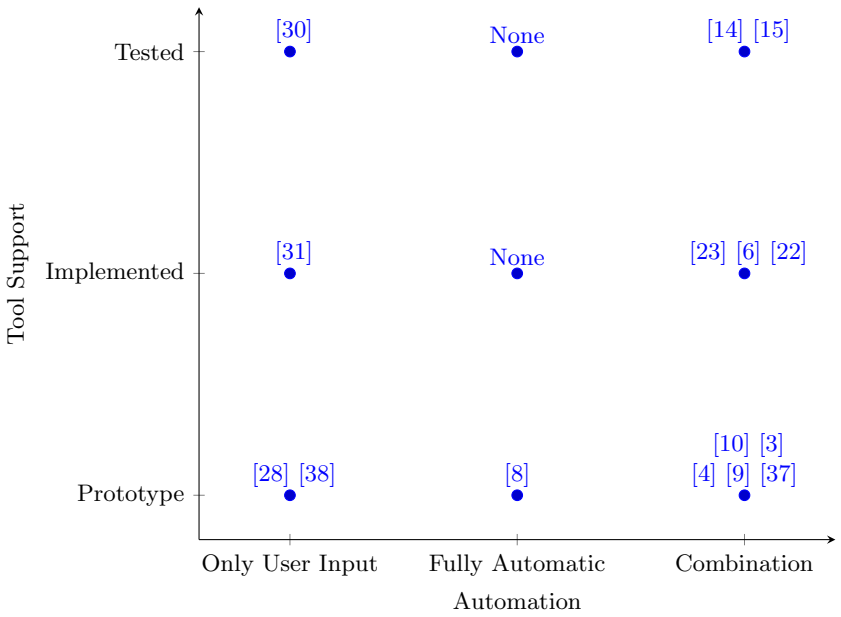

Fig. 4: Automation vs. Tool Support

In Figure 4 all solutions are presented in a matrix which uses these two criteria as axes. As can be seen in this matrix, solutions which are only based on user input already have mature tool support. As in the previous section, COPE and Epsilon Flock are both

implemented and tested on industrial-sized projects. There exists only one prototype supporting fully automatic migration of instance models, presented in [8]. From previous sections and this matrix we can conclude that this approach is not popular because it is a challenging task to completely automate the migration process: some metamodel adaptations are simply too specific to resolve automatically by a generic algorithm. Authors have therefore attempted the creation of an algorithm which can automatically detect metamodel adaptations and create a model migration transformation with, for some types of adaptations, transformation steps provided by the user. However, it remains to be investigated whether these tools are capable of supporting co-evolution in industrial-sized projects.

3.3 Algorithm vs. Tool Support

This section will discuss tool support for the different types of algorthms identified in solutions to the co-evolution problem. These algorithms have not been explicitly discussed in the previous sections, but have been mentioned a number of times. We distinguish four different algorithms. The first are manual algorithms, which require the user to manually specify the migration transformation steps (in a language like Epsilon Flock). Second are operator-based solutions, which allow the user to specify metamodel adaptations and model/transformation migration steps using coupled operators. The last two categories are used in solutions which co-evolve metamodels, models and transformations semi-automatically and are in need of a method to build a difference model. This difference model can either be constructed by comparing the two versions of the metamodel using a tool like EMF Compare [18] or by keeping track of the changes performed on the metamodel and generating a change trace.

In Figure 5 the resulting matrix is presented (note that some of the solutions are listed twice: these solutions can either use a differencing or change recording algorithm). A few interesting results can be concluded from this matrix. Firstly, as is the case in the preceding matrices as well, COPE and Epsilon Flock are the only tools that have been thorougly tested for the manual and operator based approaches, respectively. Secondly, only prototypes for the dif-

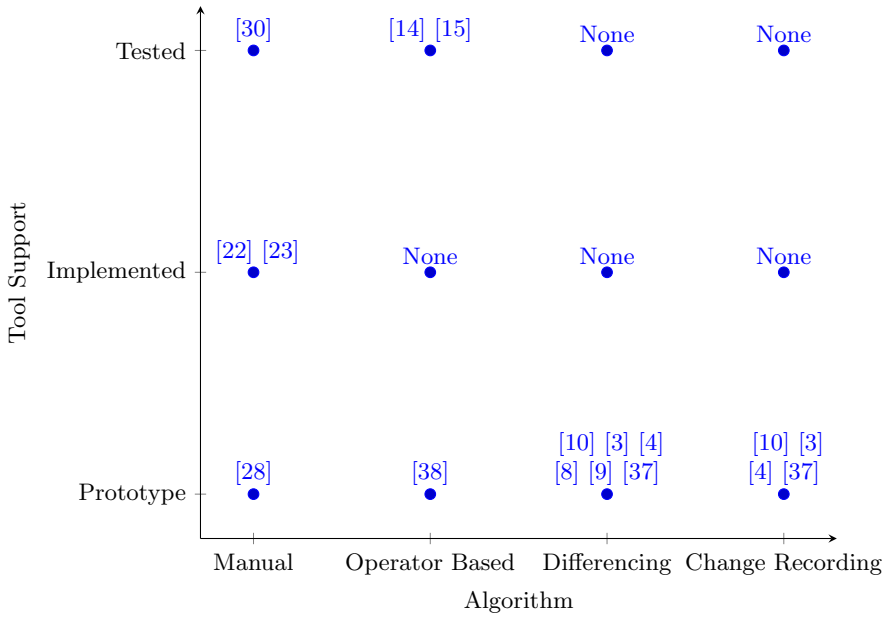

Fig. 5: Algorithm vs. Tool Support

ferencing and change recording algorithms have been created. This means these two method of co-evolving metamodel and conforming models and transformations have never been validated. Most of the approaches which use a combination of automation and user input employ a three-step algorithm: change recording/detecting, mapping of change model on model migration steps (either automatically or through user input collection) and execution of migration transformation. A tool which implements this approach and is tested on industrial-sized projects has yet to be created.

3.4 Migration Support vs. Example

Examples are often used to clarify concepts and algorithms used in a paper. In the case of metamodel and model co-evolution, examples are often given with respect to categorization of metamodel adaptations (see Section 2.1 for the different types of categorizations employed by authors) and the algorithms used by the proposed solution. We differentiate between two different types of examples: toy examples and full fledged examples. Toy examples are examples

which either only demonstrate a classification or algorithm on trivial evolution scenarios or demonstrate the funcionality of the solution on a toy language. A full fletched example demonstrates the working of an algorithm by executing it on a set of non-trivial evolution scenarios or on an industrial-sized project.

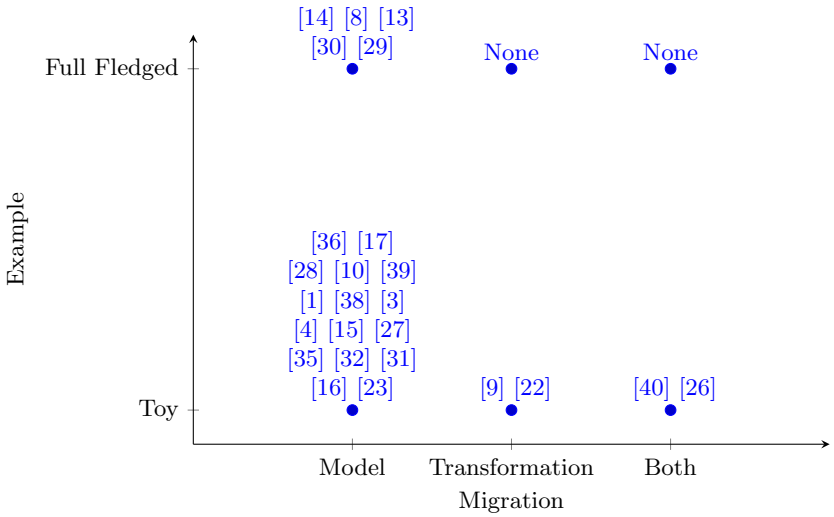

Fig. 6: Migration Support vs. Example

In Figure 6 migration support of solutions and the examples presented in the corresponding paper which demonstrate the functionality of the solution are combined into a matrix. The results are explained below, where we present each type of migration support and which papers present which type of example.

Model: In [36] an example model migration transformation is constructed using the DSL introduced in the paper. The language being evolved is used to create models in the domain of signal processing. In its original version three concepts are defined: *Ports*, *Connections* and processing blocks which are connected to each other using *Ports* and *Connections*. In the evolved version of the language *Ports* are specialized into *InputPorts* and *OutputPorts*. As a specific semantic meaning to these concepts is attached, the authors define a migration transformation which maintains the semantic meaning of models af-

ter migration.
[17] presents a classification of metamodel adaptations and presents model migration algorithms for each identified class of changes. Example metamodel adaptations are presented visually. However, the examples given neither form a complete set of metamodel adpatations nor have they been applied to a real-life project.
In [28] a toy example is presented to demonstrate the functionality of *Lever*. It concerns an expression language that translates sums to stack machine code. The authors transform the language from infix to postfix notation by specifying a metamodel evolution which can automatically migrate instance models as well.
[10] introduces a change classification and algorithm based on this classification scheme. To demonstrate their approach, the authors present an example metamodel of a file system. This metamodel is evolved by performing certain adaptations such as renaming, subclassing and creating or moving of containment references. To capture the semantic intention behind these changes, the method proposed in the paper constructs migration transformations which are capable of maintaining this semantic information.
In [39] an extensive categorization of metamodel adaptation is presented. For each category, a set of example adaptations are modelled as coupled operators using QVT Relations. An example Petri Net metamodel evolution is presented, clarifying the general concept of metamodel and model co-evolution.
In [38] the concept of metamodel and model co-adaptation is applied to the domain of data modelling. A set of metamodel adaptations is presented and corresponding model migration steps are derived. Then, these examples are used to construct a general architecture which consists of an automatically constructed DSL for the construction of metamodel adaptations. The examples are then represented using this general architecture to show its functionality.
The authors in [3] validate their approach by applying it on an evolving Petri Net metamodel. The metamodel is evolved in two steps using a set of metamodel adaptations such as 'restrict metaproperty', 'introduce superclass' and 'introduce property'. Only fragments of the resulting model migration transformations are shown in the paper. The same Petri Net metamodel is used by the authors in [4] which explains how to deal with dependent metamodel adaptations.

COPE has been tested on a large number of evolution scenarios. As it was not possible to deal with model-specific adaptations in the first version of COPE, the authors present the needed constructs in [15]. A toy example is presented to demonstrate these constructs which evolves an automata metamodel to support initial states. This is a model-specific adpatation as the modeller has to choose an initial state for each evolved model.
The MCL is used in [27] to co-evolve metamodel and models. A set of example MCL patterns is presented for metamodel adaptations such as the introduction of subclasses and the changing of containment hierarchy.
In [35] a theoretical overview of domain-specific languages is presented. The authors use their definitions to present the different types of evolution that can be performed on a system created using a DSL. An example language is used to clarify the concepts introduced and to argue which metamodel adaptations can be trivially mapped to model migrations and which are in need of user intervention.
In [32], Rose et al. compare different approaches to model migration. Several examples are given to clarify migration scenarios for the different approaches (the authors distinguish between manual specification, operator based and metamodel matching approaches). Rose et al. introduce a model migration approach in [31] which is based on binding models to a generic metamodel, after which inconsistencies are presented to the user in a resolution environment. A toy example is presented to clarify the approach and show the inconsistency report generated by the tool.
In [16], Herrmannsdoerfer presents an extensive catalog of reusable coupled operators. A number of these coupled operators are clarified by presenting example evolution scenarios.
In [23] a running metamodel evolution scenario is used throughout the paper to clarify the approach of using inplace transformation languages to specify model co-evolution. The example is used at every step of the process: first, the original and evoled metamodel are used to create a merged metamodel. Then, the co-evolution rules are expressed as graph transformations, which make use of this merged metamodel. These graph transformations are then mapped onto ATL transformation rules.

COPE, as has been mentioned in the previous sections, has been tested thoroughly. In the paper which introduces the tool [14], the authors already demonstrate the full functionality by presenting different adaptation scenarios of a simple domain-specific language. These examples are complete in the sense that every supported type of operation (reusable and user-defined) is explained in these examples. Next to these small examples, a case study is performed on two EMF-based metamodels: one is part of the Graphical Modelling Framework (GMF), the other is part of the Palladio Component Model (PCM). Both of these metamodels already have a rich evolution history and this history has been recreated through the use of COPE. The results have been analyzed in the paper to test the viability of the operator-based approach. A similar case study has been performed in [13] on two industrial-sized metamodels. The adaptations performed on these metamodels have been analyzed and classified according to the classification of Herrmannsdoerfer (see Section 2.1). From these results, a number of requirements for automated coupled evolution are formulated, which reappear in the subsequent papers on COPE.

In [8] a method is proposed based on precise detection of metamodel adaptations through the use of heuristics. To clarify the approach, a running example is used based on an evolution scenario of a Petri Net metamodel. The approach is validated by implementing a prototype tool and testing it on six versions of a Petri Net model and eight versions of the Netbeans Java metamodel. The matching strategies composed of a set of heuristics for both metamodels are presented and the results are analyzed to demonstrate that the approach detects most metamodel adaptations correctly.

In [30] Epsilon Flock is introduced. Its functionality is demonstrated using two example metamodel adaptation scenarios. The first uses a Petri Net metamodel which is included as a comparison to other approaches. The other uses the UML metamodel and the adaptations performed from UML 1.5 to UML 2.0. Flock also appears in [29], where it is compared to COPE, amongst others. The tools are compared by first applying them on an example migration of a Petri Nets metamodel. Then they are applied to the larger example of evolution of the GMF metamodel. It shows both COPE and Epsilon Flock can deal with the complex evolution of large metamodels.

Transformation: As for the two papers that introduce methods for transformation migration, they both clarify their solutions by presenting toy examples.
In [9] an example transformation is described by a number of ATL rules. The transformation has as its domain an exam language which describes an exam by a list of questions and as its image an MVC language. When a metamodel adaptation is performed, the transformation is migrated by a higher-order transformation. Missing semantic information may need to be provided manually by the developer.
In [22] the MCL is used to migrate transformations. A case study is presented for the domain of hierarchical signal flows, which are mapped onto an actor-based language without hierarchy by a transformation. The domain metamodel is evolved and transformation instances co-evolved by specifying a mapping in MCL. Adaptations that can be automatically mapped to model migrations are generated and missing semantic information should be provided by the developer where necessary.

Both: In [40] both model and interpreter evolution is discussed. An architecture is presented which should be able to migrate models and interpeters starting from a metamodel evolution specification. A few example metamodel adaptations are presented together with their remedial actions in models and interpreters. In all of the cases, a trivial search/replace algorithm is sufficient to migrate these instances.
In [26] an example language is used throughout the paper. This DSL is used to specify railroad networks, which are mapped onto Petri Nets by a transformation which attaches semantics to the constructed models. The language is used to clarify concepts introduced in the paper and show limitations of transformation migration (which is called the projection problem). An example evolution is performed on the metamodel of the language whereby a migration pipeline is built for the evolution of instance models and transformations created using the initial version of the language. In the evolution scenario, only a few operations are performed on the metamodel and as such it is classified as a toy example.

4 Conclusion

This paper has discussed the currently available literature on metamodel, model and transformation co-evolution using serval criteria: change classification and support, migration support, automatibility and completeness, intention preservation and tool support. Then, these criteria were combined into a number of matrices which lead to directions for future work in this area. The following was concluded:

- Additional research is required to construct tools which support transformation migration and tools which support both model and transformation migration.
- Validation studies are needed to test the correct functioning of tools that (semi-)automatically migrate instance models or transformations through the construction of a difference model, as these tools have not been tested on industrial-sized projects.
- Full fletched examples for solutions that migrate transformations or migrate both models and transformations have to be presented. Currently, no large case studies have been performed that validate these solutions.
- The ability to support intention preservation and continuity has to be further investigated in future tool implementations.
- A standard case study should be created which can be used to test the functionality of a proposed approach. This case study should include a complex metamodel evolution scenario which consists of all possible types of changes.
- It is not possible to construct a fully automatic solution for co-evolving metamodels, models and transformations due to the fact that the intention of the language developer has to be captured in some way. Researchers should focus on semi-automatic methods instead.

Further research into these fields should increase the general level of understanding of the subject and lead to ful support of the co-evolution of metamodels and their related artefacts, allowing the adoption of MDE as a mainstream software development method.

References

1. Steffen Becker, Thomas Goldschmidt, Boris Gruschko, and Heiko Koziolek. A process model and classification scheme for semi-automatic meta-model evolution. In *Proc. 1st Workshop MDD, SOA und IT-Management (MSI'07)*. GiTO-Verlag, 2007.
2. Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture*, pages 33–46. 2005.
3. Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.
4. Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing dependent changes in coupled evolution. In Richard F. Paige, editor, *Proc. 2nd International Conference on Model Transformation (ICMT'09)*, volume 5563 of *LNCS*, pages 35–51, Zurich, Switzerland, 2009. Springer.
5. M. D. Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. Amw: A generic model weaver. *Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles*, 2005.
6. Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. Automated co-evolution of gmf editor models. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 143–162, Berlin, Heidelberg, 2011. Springer-Verlag.
7. Peter Fr¨hlich and Johannes Link. Automated test case generation from dynamic models. In Elisa Bertino, editor, *ECOOP 2000 Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 472–491. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-45102-1_23.
8. Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 34–49, Berlin, Heidelberg, 2009. Springer-Verlag.
9. Jokin Garcia and Oscar Daz. Adaptation of transformations to metamodel changes. *Library*, 2:1–9, 2010.
10. Boris Gruschko, Dimitrios S. Kolovos, and Richard F. Paige. Towards synchronizing models with evolving metamodels. In *In Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.
11. David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, October 2004.
12. Zef Hemel, Lennart Kats, and Eelco Visser. Code generation by model transformation. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 183–198. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69927-9_13.
13. Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Automatability of coupled evolution of metamodels and models in practice. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 645–659, Berlin, Heidelberg, 2008. Springer-Verlag.

14. Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope - automating coupled evolution of metamodels and models. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 52–76, Berlin, Heidelberg, 2009. Springer-Verlag.
15. Markus Herrmannsdoerfer and Daniel Ratiu. Limitations of automating model migration in response to metamodel adaptation. In Sudipto Ghosh, editor, *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2009.
16. Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 163–182, Berlin, Heidelberg, 2011. Springer-Verlag.
17. Joachim Hssler, Michael Soden, and Hajo Eichler. *Coevolution of Models, Metamodels and Transformations*, chapter Coevolution of Models, Metamodels and Transformations. Wissenschaft und Technik Verlag, Berlin, 2005.
18. Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.
19. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon object language (eol). In *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ECMDA-FA'06, pages 128–142, Berlin, Heidelberg, 2006. Springer-Verlag.
20. D.S Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, United Kingdom, 2009.
21. Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5:369–385, 2006. 10.1007/s10270-006-0017-9.
22. Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, and Gabor Karsai. A novel approach to semi-automated evolution of dsml model transformation. In *Proceedings of the Second international conference on Software Language Engineering*, SLE'09, pages 23–41, Berlin, Heidelberg, 2010. Springer-Verlag.
23. J. Schönböck W. Retschitzegger W. Schwinger G. Kappel M. Wimmer, A. Kusel. On using inplace transformations for model co-evolution. In *Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL 2010)*, 2010.
24. Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformation. In *Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl*. Electronic, 2005.
25. Tom Mens, Michel Wermelinger, Stphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Proc. 8th Intl Workshop on Principles of Software Evolution*, pages 13–22. IEEE, September 2005.
26. Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Sci. Comput. Program.*, 76(12):1223–1246, December 2011.
27. Anantha Narayanan, Tihamer Levendovszky, Daniel Balasubramanian, and Gabor Karsai. Automatic domain model migration to manage metamodel evolution. In *Proceedings of the 12th International Conference on Model Driven Engineering*

Languages and Systems, MODELS '09, pages 706–711, Berlin, Heidelberg, 2009. Springer-Verlag.

28. Markus Pizka and Elmar Jurgens. Automating language evolution. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, TASE '07, pages 305–315, Washington, DC, USA, 2007. IEEE Computer Society.
29. Louis Rose, Markus Herrmannsdoerfer, James Williams, Dimitrios Kolovos, Kelly Garcés, Richard Paige, and Fiona Polack. A comparison of model migration tools. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, chapter 5, pages 61–75–75. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.
30. Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with epsilon flock. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 184–198, Berlin, Heidelberg, 2010. Springer-Verlag.
31. Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. *Automated Software Engineering, International Conference on*, 0:545–549, 2009.
32. Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. An analysis of approaches to model migration. In *Proc. Models and Evolution (MoDSE-MCCM) Workshop, 12th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems*, October 2009.
33. Ákos Schmidt and Dániel Varró. Checkvml: A tool for model checking visual modeling languages. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 92–95. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-45221-8_8.
34. Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39:25–31, 2006.
35. Jonathan Sprinkle, Jeffrey Gray, and Marjan Mernik. Fundamental limitations in domain-specific language evolution. *IEEE Transactions on Software Engineering*, 2009.
36. Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, June 2004.
37. Mark Van Den Brand, Zvezdan Protić, and Tom Verhoeff. A generic solution for syntax-driven model co-evolution. In *Proceedings of the 49th international conference on Objects, models, components, patterns*, TOOLS'11, pages 36–51, Berlin, Heidelberg, 2011. Springer-Verlag.
38. Sander Vermolen and Eelco Visser. Heterogeneous coupled evolution of software languages. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 630–644, Berlin, Heidelberg, 2008. Springer-Verlag.
39. Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In Erik Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer-Verlag, July 2007.
40. Jing Zhang, Jeff Gray, and Yuehua Lin. A generative approach to model interpreter evolution.