The Journal of Systems and Software 212 (2024) 112012

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Supporting reusable model migration with Edelta☆

Lorenzo Bettini [a],*, Amleto Di Salle [b], Ludovico Iovino [b], Alfonso Pierantonio [c]

[a] *Università degli studi di Firenze, 50134 Firenze, Italy*
[b] *Gran Sasso Science Institute, 67100 L'Aquila, Italy*
[c] *Università degli studi dell'Aquila, 67100 L'Aquila, Italy*

ARTICLE INFO

Keywords:
MDE
Coupled evolution
Co-evolution
Migration
Adaptation

ABSTRACT

In Model-Driven Engineering, metamodels define the vocabulary of concepts and relations that designers use to define a wide range of artifacts, including models, transformations, and editors. Therefore, whenever a metamodel undergoes modifications, the depending artifacts may no longer be valid, and consistency needs to be repaired through coupled evolution techniques. While several approaches have been proposed over the last decades, they are artifact- and domain-specific and do not facilitate the reuse of migration strategies. Indeed, migration strategies are often hard-coded for a given project in a specific domain. In this paper, we propose the novel concept of migration patterns to leverage reuse across different domains and projects. The approach extends the existing Edelta framework and has been evaluated by considering several case studies identified in a systematic literature review.

1. Introduction

In Model-Driven Engineering (Schmidt, 2006) (MDE), models are elevated to first-class status and can be manipulated through automated transformations. Models are defined based on the concepts and relationships specified in metamodels, which are the fundamental building blocks for creating artifacts such as models, transformations, and editors.

When a metamodel evolves, however, existing artifacts may become corrupted, and the consistency between them must be restored using the techniques of *coupled-evolution* (co-evolution). In coupled evolution, each change in the metamodel can impact the involved artifacts differently, and existing works in the literature classify them based on the impact as non-breaking, breaking and resolvable, or unresolvable changes (Cicchetti et al., 2008a). Multiple approaches have been proposed in the last few years to deal with the co-evolution problem, especially in the model/metamodel setting, but also considering transformations or code-generator adaptation (artifact-specific approaches).

Approaches for re-establishing conformance can be manual, where transformations are manually crafted to transform existing models to comply with the new metamodel variant (Yu and Berg, 2015). Alternatively, automated approaches can be based on inference approaches where migration actions can be derived by comparing the original with the evolved metamodel. Automated tools, such as Epsilon Flock (Rose et al., 2010b) or COPE (Herrmannsdoerfer et al., 2009), are *operator-based* tools, based on DSLs for expressing the migration strategies. Metamodel changes are often defined as patterns that recur in different metamodels. Existing approaches for dealing with model migration offer a way of specifying migration strategies for the affected case studies. This means that a modeler needs to specify the migration strategies that act on the affected models by considering the evolution of the specific metamodel to which these models conform. Even though the migration strategies are specified with dedicated tools, supported by textual DSLs or graphical rules, the level of reuse is quite limited. Indeed, for each evolving case study, the modelers need to specify the migration rules from scratch and apply the migration for each model to be migrated.

In this paper, we propose a novel classification for migration strategies based on the concept of *migration pattern*, which is based on the recurrence of the migration strategies across different scenarios. We also, hypothesize that recurring migration strategies can be specified as migration patterns that can be applied to cross-domain case studies.

This opens up a possible classification of *domain-specific* and *domain-independent* migration patterns. We confirm this hypothesis with a light systematic literature review, on whose results we base the Edelta (Bettini et al., 2017) evaluation shown in this paper. We previously proposed Edelta as a metamodel evolution framework, and in this paper, we present an extension for supporting the specification of migration

☆ Editor: Laurence Duchien.
* Corresponding author.
E-mail addresses: lorenzo.bettini@unifi.it (L. Bettini), amleto.disalle@gssi.it (A. Di Salle), ludovico.iovino@gssi.it (L. Iovino), alfonso.pierantonio@univaq.it (A. Pierantonio).

https://doi.org/10.1016/j.jss.2024.112012
Received 8 March 2023; Received in revised form 8 February 2024; Accepted 21 February 2024
Available online 28 February 2024

patterns. We propose a catalog of migration patterns specified in an Edelta library that can be used to support the migration of cross-domain case studies. The modeler can simply import our library and use the existing refactorings for migrating their projects.

We evaluated the novel extension of Edelta with selected case studies extracted from the literature for answering the research question and confirmed that we are able to specify reusable migration patterns with different levels of automation.

Structure of the paper. Section 2 describes the background concepts and proposes an SLR to deeply investigate and characterize the topic; Section 2 also shows some motivating examples, based on the result of the analysis; Section 3 describes the novel approach based on the Edelta extension; Section 4 demonstrates the proposed approach on a set of running examples; Section 5 evaluates the proposed approach and Section 6 discusses the highlights. The threats to validity are explored in Section 7. Section 8 presents related works and Section 9 draws the conclusion and describes the future work.

2. Model/metamodel co-evolution

This section first introduces background concepts about model and metamodel co-evolution and new definitions we will use throughout the paper. Then we use a light SLR to analyze how this problem has been treated in the literature.

2.1. Background

Metamodels can evolve, like many software artifacts (Di Ruscio et al., 2011). When a metamodel evolves, existing models, previously compliant to the metamodel, may become corrupted and must be migrated to reestablish conformance (Williams et al., 2012). A metamodel can evolve in multiple ways to consider new emerging requirements or to fix existing inaccurate modeling concepts (Iovino et al., 2012). The activity of restoring the lost conformance relationship between models and their metamodel is called *coupled-evolution* (or *co-evolution*). Multiple metamodel evolution patterns can be applied, from simple (also called atomic) to complex (Hebig et al., 2017), from additive to subtractive changes. These changes can be categorized w.r.t. the metamodel/model co-evolution, based on the evolution pattern's effect on the model. These changes have been previously categorized as: *non-breaking* (NBC), *breaking and resolvable* (BRC) or *breaking unresolvable* (BUC) (Cicchetti et al., 2008b). The first category includes the metamodel changes not impacting the model conformance. An example is the extraction of a superclass. By definition, if we extract a superclass from a commonly defined list of features, the existing model instances of the (new) subclasses in the evolved metamodel are not affected by the change. Indeed, this change is transparent to the existing models, which will remain valid. BRCs are changes applied to the metamodel that can trigger an automatic model migration strategy, often unique. An example is renaming a metaclass and migrating the model by retyping the existing instances of the renamed type with the new type. BUCs are changes that can be faced by using heuristics encoded in the resolution strategy. However, they typically require user intervention: the user is requested to specify the actions with a migration engine or other tools.

Multiple solutions (Paternostro and Hussey, 2006; Taentzer et al., 2013; Cicchetti et al., 2008a; Garcés et al., 2009) and tools (Rose et al., 2010b; Herrmannsdoerfer et al., 2009) have been proposed to deal with the problem of model–metamodel co-evolution in the last decade. Usually, these approaches encode predetermined migration strategies.

Migration strategy. A migration strategy M_s is a specific strategy of migration of an affected model in response to an evolution scenario applied to the corresponding metamodel *MM*.

All these approaches share that in every technique, the migration is specified as domain-dependent at the specific metamodel level. Indeed, if one has a metamodel evolution pattern, the migration strategy is specified for each evolving metamodel based on the fact that the resolution can be specific to the evolving domain.

Based on this evidence we formulated a hypothesis:

Hypothesis 1. There could be cases in which the migration strategy may be re-used across metamodels.

To better frame the hypothesis, we also define the following concepts.

Evolution pattern. An evolution pattern E_p is a recurring evolution scenario applied to a metamodel.

Examples of evolution patterns have been described in Herrmannsdoerfer et al. (2010) and in Bettini et al. (2022a) and include *Introduce subclasses*, where a metaclass of a metamodel is modified to become abstract and subclasses are introduced to specialize it, or *Merge attributes*, where a metaclass in a metamodel has attributes that are merged in an evolved version of the metamodel. All these patterns identify recurrent metamodel evolution scenarios and do not consider what happens at the other defined artifacts.

Migration Pattern. We define a (co-evolution) migration pattern[1] M_p a recurring co-evolution pair, defined as: $\langle E_p, M_s \rangle$, where E_p identifies a specific metamodel evolution pattern, e.g., renaming a metaclass, and M_s identifies a possible migration strategy, e.g., retyping all the instances of the renamed metaclass.

It is worth noting that migration strategies are not unique and univocal (Di Ruscio et al., 2016) and can include multiple options depending on multiple constraints or application and quality factors. A co-evolution pattern then considers the evolution of a metamodel and the corresponding effect on the models, as well as the migration to be applied to restore the conformance. With current co-evolution approaches (see, e.g., Hebig et al. (2017)), even when metamodels formalize the same application domain, e.g., financial applications or web applications, or when metamodels share structural or semantic similarities as, e.g., in activity diagrams and statecharts, the expert dealing with the co-evolution process is nevertheless forced to re-write or redefine the co-evolution strategy with additional effort and limited re-use.

For this reason, metamodel evolution scenarios induce required migrations that can be not only classified w.r.t. the effect on the models and automation of the resolution strategy but also the possible level of reuse of a determined migration strategy, i.e., domain-specific or -independent. Based on these premises we can define a **Migration Pattern** as:

Domain-specific.[2] A domain-specific migration pattern is a migration pattern that can be applied to models that conform to a given domain.

Domain-independent. A domain-independent migration pattern is a migration pattern that can be applied to generic co-evolution patterns independently from the metamodel representing a specific domain, and thus it can be applied across multiple domains (cross-domain).

[1] We use the terms migration pattern and co-evolution pattern interchangeably even though "migration" is often used only to refer to model/metamodel co-evolution.

[2] We can use the terms *domain-specific* and *domain-dependent* as interchangeable.

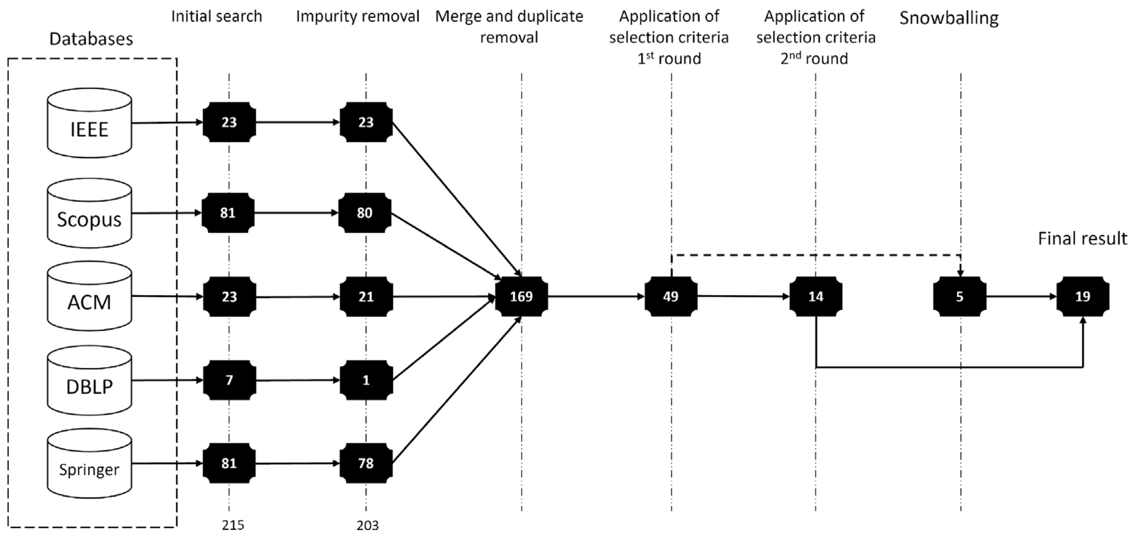


Fig. 1. The process of searching and selecting to identify relevant studies.

2.2. Literature analysis

Based on the above hypothesis, we performed a light systematic literature review (SLR) (Kitchenham et al., 2009) with the intent of answering the following research questions:

RQ1: What migration patterns are supported by the existing co-evolution approaches?
RQ2: Are migration patterns applied (and shared) across multiple domains?

2.2.1. Search process

The search strategy has been defined by answering the following four W-questions (Zhang et al., 2011) –*Which?*, *Where?*, *What?*, and *When?*.

Which? Automatic and manual searches were performed for relevant papers from conferences and journals.
Where? The defined search strategy has been applied to digital libraries available online, in which we selected the following five to run the automated search to guarantee a certain level of quality of relevant papers: *IEEE Xplore*,[3] *Scopus*,[4] *Association for Computing Machinery (ACM)*,[5] *dblp computer science bibliography*,[6] and *Springer Link*.[7]
What? We defined a query string to be executed on different digital resources to collect papers. To this end, the *query string* that we conceived is the following[8]:
Search String: `(Title:(‘‘model migration’’) OR Title:` `(‘‘model co-evolution’’) OR Title:(‘‘model adaptation’’))` `AND (Fulltext:(MDE) OR Fulltext:(Model Driven Engineering))`
When? We considered papers that have been published in the period 2005–2023.

[3] https://ieeexplore.ieee.org/Xplore/home.jsp
[4] https://bit.ly/3m4T74d
[5] https://www.acm.org/
[6] https://dblp.uni-trier.de/
[7] https://link.springer.com/
[8] We reported one of the strings executed on one of the digital libraries, the others are equivalent but with different syntax and operators.

Springer Link’s search engine does not allow a combined full-text or title search. Therefore, we limit the search to article titles, i.e., “model migration”, “model co-evolution”, and “model adaptation”. As for Scopus, it does not permit full-text searching. Therefore, we limit the search to titles, keywords, and abstracts for “MDE” and “model driven engineering” keywords.

2.2.2. Selection process

Fig. 1 shows the process of searching and selecting relevant studies from the defined five databases. In the initial search, 215 studies have been identified. Then, the following selection process steps are executed to obtain the relevant studies.

▹ **Impurity removal:** In this step, we excluded non-research papers like books, reports, theses, and articles that were unavailable or not published in a journal or presented at a conference. After executing the impurity removal step, we obtained 203 studies, including 21 for the ACM, 1 for the DBLP, and 78 for the Springer databases.

▹ **Merge and duplicate removal:** We combined papers from the five sources into a single corpus and removed duplicates, resulting in 169 studies.

▹ **Application of selection criteria 1st round:** During the first phase of study selection, the second and third authors examined the titles and abstracts of the studies and identified potential studies based on the following selection criteria. The authors included studies if they could not judge them by only reading the titles and abstracts. The selection criteria are organized in inclusion criteria (I):

I1 Studies subject to peer review;
I2 Studies that propose a tool-supported approach;
I3 Studies that clearly mention or propose migration patterns, i.e., evolution of the metamodel and corresponding migration strategy of the model;

whereas the exclusion criteria (E) of our study are reported in the following:

E1 Papers that are not written in English;
E2 Short papers, tutorial slides, or technical reports;
E3 Studies or tools working on artifacts that are not modeling artifacts, e.g., code adaptation.
E4 Studies not reporting clear examples of the application of the migration pattern that do not permit the formalization of a clear migration pattern.

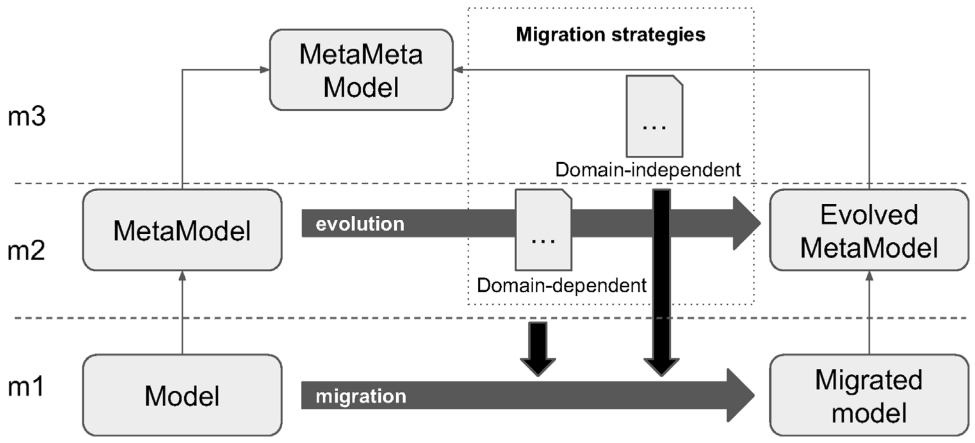


Fig. 2. Model/metamodel co-evolution.

After the first round, 49 studies have been identified.

▷ **Application of selection criteria 2nd round:** The second and third authors independently read the full papers coming from the first round. They made the final decision on whether a study should be selected or not. If a study was considered controversial, all the authors discussed it and agreed on whether it should be included. At the end of the second round, 14 studies have been selected.

▷ **Snowballing:** We complemented the selection process using the snowballing technique (Wohlin, 2014). In particular, we only conducted the backward snowballing activity where the third author read the references of the studies extracted in the first round of study selection. In the snowballing activity, we extracted 5 studies.

At the end of the searching and selecting process, 19 studies have been identified.[9]

▷ **Data extraction:** The second and third authors extracted information about the selected primary studies to answer the RQs. In particular, concerning the RQ1, they collected the co-evolution patterns indicated and described in the articles using and adding, when not present, the classification on the effect of the co-evolution on models defined in Cicchetti et al. (2008b). Regarding RQ2, they extracted the examples together with their co-evolution migration patterns. Tables 1 and 2 (in Section 5) and Fig. 2 are the results of the data extraction activity.

2.3. *Analysis of the results*

Table 1 summarizes the results of the data extraction phase by indicating the patterns of co-evolution divided by atomic or complex and non-breaking (NBC), breaking and resolvable (BRC) or breaking unresolvable (BUC) (Cicchetti et al., 2008b). The table also represents the main result in response to **RQ1**.

All the approaches in Table 1 specify and manage migration strategies as domain-specific, meaning that for each evolution scenario, we need to implement a migration strategy for the treated domain. Note that the last row in Table 1 reports a paper exploring the patterns, but with exemplary scenarios, e.g., metaclass A-B-C-D, so the patterns are not referable to any realistic example, which is why it is not included in Table 2. This general approach to managing co-evolution is depicted in Fig. 2, where we also reported the MOF four-layered architecture (Cadavid et al., 2012) to highlight that domain-dependent migrations are expressed at m2 level since concepts of the specific metamodel are needed to express co-evolution patterns.

Therefore, we may specify meta-migration strategies going beyond the specific metamodel and express them as domain-independent. This meta-level of expression is conceptually close to expressing general migration strategies at the meta-metamodel level, that in EMF (Steinberg et al., 2008) for instance, would correspond to Ecore.

In the remainder of this section, we show some examples of metamodel evolution and corresponding model migration patterns, extracted from Table 1, by using an excerpt of metamodel/model co-evolution examples.[10] Moreover, since we wanted to have statistical applicability of the identified patterns, we also inspected the repository presented in Barriga et al. (2020), looking for similar examples of the metamodels in the Table.

In the following, we show how we analyzed the extracted works. We use this knowledge later in Section 5 to fill Table 2 to answer **RQ2**.

2.3.1. *Introduction of subclasses*

The metamodel excerpt in Fig. 3 represents one of the existing metamodels in the literature, "engineering state charts". Some of the existing metamodels, w.r.t. to the version reported in Fig. 3 (left), report additional subclasses of the metaclass State, i.e., InitialState, FinalState, as in Fig. 3 (right).

This can be considered an evolution scenario, as proposed in Kessentini et al. (2016), in which an existing concrete metaclass in the initial metamodel evolves to abstract and new subclasses are introduced in the evolved metamodel. If models conform to the initial metamodel exist, the declared instances of Node need to be migrated to the newly introduced subclasses, being Node abstract. This change is classified as BUC since the migrator needs information about how to retype the existing instances according to the newly introduced subclasses. Generally, some user intervention is needed in these cases, but there are cases in which a different migration strategy can be applied. For example, if we introduce only a single subclass, the change becomes BRC since all the instances can be retyped automatically (Mig_1). Let us consider the specific case in Fig. 3. We could apply a strategy in which the first node of the nodes list in Model can be automatically retyped to an InitialNode, the last one can be retyped to FinalState and the rest can be retyped to State (Mig_2). In the repository of metamodels published in Barriga et al. (2020), there are 28 metamodels classified as Statecharts. In some of these cases, if the introduction of subclasses is applied, it may be related to the exact pattern (Mig_2) we represented in Fig. 3, enforcing the need for general, reusable, and flexible model migration strategy definition.

Moreover, there are metamodels in which specialization of the introduction of subclasses pattern occurs. This case occurs when an enumeration is converted into subclasses of the metaclass in which the

[9] At the following GitHub repository https://github.com/amletodisalle/JSS_edelta-migration-SLR a spreadsheet with the information about the extracted papers can be found.

[10] Some examples are directly extracted from the papers in Table 1, others come from a public repository published in Barriga et al. (2020).

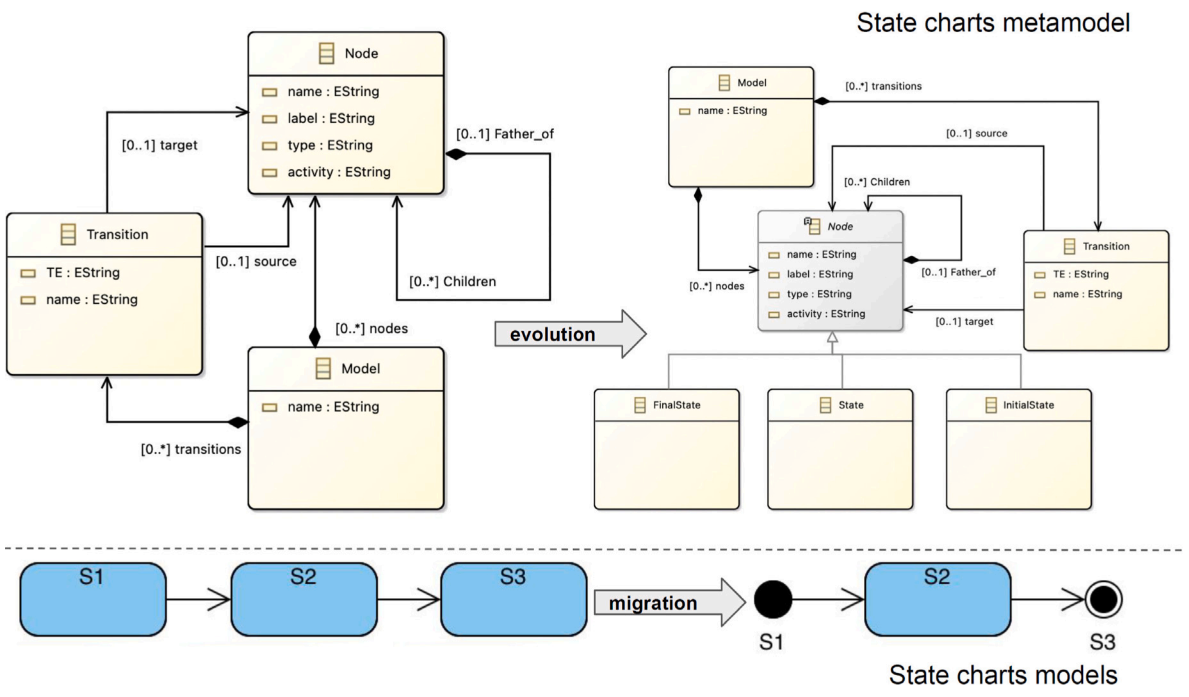


Fig. 3. Mig_2: Example of introduction of subclasses migration (Kessentini et al., 2016).

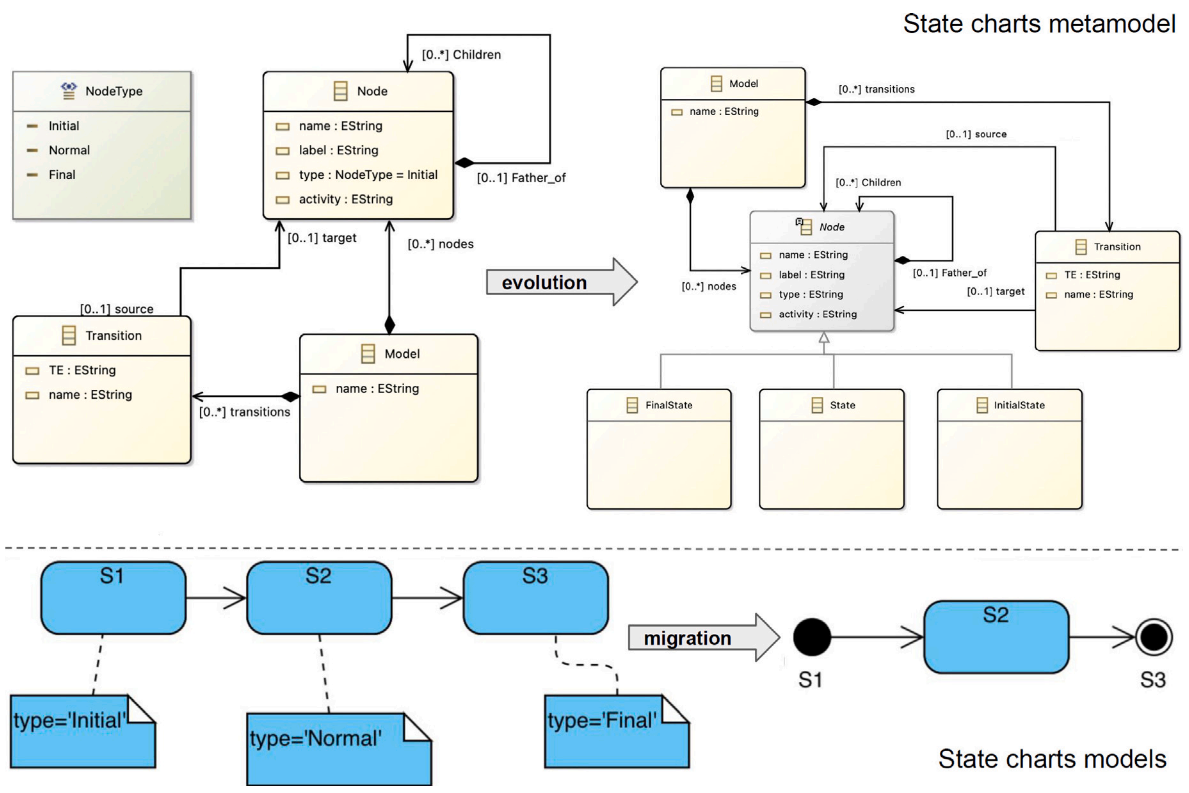


Fig. 4. Mig_3: Example of enumeration to subclasses migration (Rutle et al., 2020).

Table 1
Literature analysis and extracted patterns.

Examples	co-evolution patterns																
	Atomic									Complex							
	BUC	BRC	BRC	BRC	BUC	BUC	BRC	BRC	BRC	BRC	NBC	BRC	BUC	BRC	BRC	BRC	BRC
	Add/Delete attribute	Rename metaclass	Move Reference	Delete metaclass	Restrict/Enlarge multiplicity	Add reference	Rename attribute/reference	Delete reference	Move/Push down Attribute	Extract metaclass	Extract super metaclass	Merge/Split attribute	Introduction of subclasses	Collapse Hierarchy	Inline metaclass	Merge metaclasses	Split reference
Rose et al. (2010a) Rose et al. (2010b)	•									•	•						
Cicchetti et al. (2008a)[a]										•							
	•	•									•						
Taentzer et al. (2012)[b]	•									•	•						
										•							
Di Ruscio et al. (2012) Wagelaar et al. (2012)	•	•								•	•						
Krause et al. (2013)	•								•	•			•				
Anguel et al. (2014)										•							
Kessentini et al. (2016)													•				
García et al. (2012)	•											•	•				
Rutle et al. (2020)		•	•									•	•	•	•		
Garcés et al. (2014)	•			•						•	•						
Demuth et al. (2016)					•	•											
Taentzer et al. (2013)					•	•							•				
Rose et al. (2014a)		•					•						•	•	•	•	
Kessentini and Alizadeh (2020)			•						•				•				
Di Ruscio et al. (2016)					•				•				•	•			
Rose et al. (2014b)			•										•				
Rose et al. (2014b)	•	•				•	•	•								•	•
Herrmannsdoerfer et al. (2010)	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•

[a] This example reports two evolution steps: MM1–>MM2–>MM3.
[b] This example reports two evolution steps: E0–>E1–>E2.

attribute of the enumeration type is defined (Rutle et al., 2020). This case is reported in Fig. 4 where the Node has an attribute called type that can be set to "Initial", "Normal" or "Final". This attribute can be used to automate the migration process and apply a retyping of the existing instances based on this attribute (Mig_3) (see Fig. 4 for the final result).

For these three cases of the same evolution/migration patterns, even if they vary only slightly, the migration strategies are usually written considering the specific domain, requiring a new migration program every time.

To summarize, concerning the identified migration strategies for this evolution pattern:

– Mig_1 is not *domain-specific*, and it can be applied every time that condition is verified in the metamodel, making the evolution a BRC.
– Mig_2 is a *domain-specific* strategy for the statecharts (and maybe other related domains), and it can be reused for most of the existing metamodels classified as 'statecharts', which introduce those subclasses in the evolved scenarios.
– Mig_3 is a specific case of the pattern, and it is a BRC and *domain-independent* since it can be applied to all the cases when this evolution pattern occurs in the metamodel.

2.3.2. Merge and split attributes

The evolution patterns treated in this paragraph deal with merging or splitting existing attributes in the metaclasses of the metamodel. The merge attributes evolution is a specialization of the *merge features* reported in Herrmannsdoerfer et al. (2010), and it is applied when multiple attributes are merged into a single one. The corresponding models become invalid since they refer to not existing attributes.

Fig. 5 reports an example of this pattern applied to the AddressBook metamodel shown in Rutle et al. (2020). In this metamodel, in the metaclass Person, the attributes firstname and lastname are merged into a single attribute name. The existing instances of Person in the models have to be migrated.[11] The migration strategy applied is quite

[11] We reported only the model excerpt touched by this evolution pattern to make the example readable.

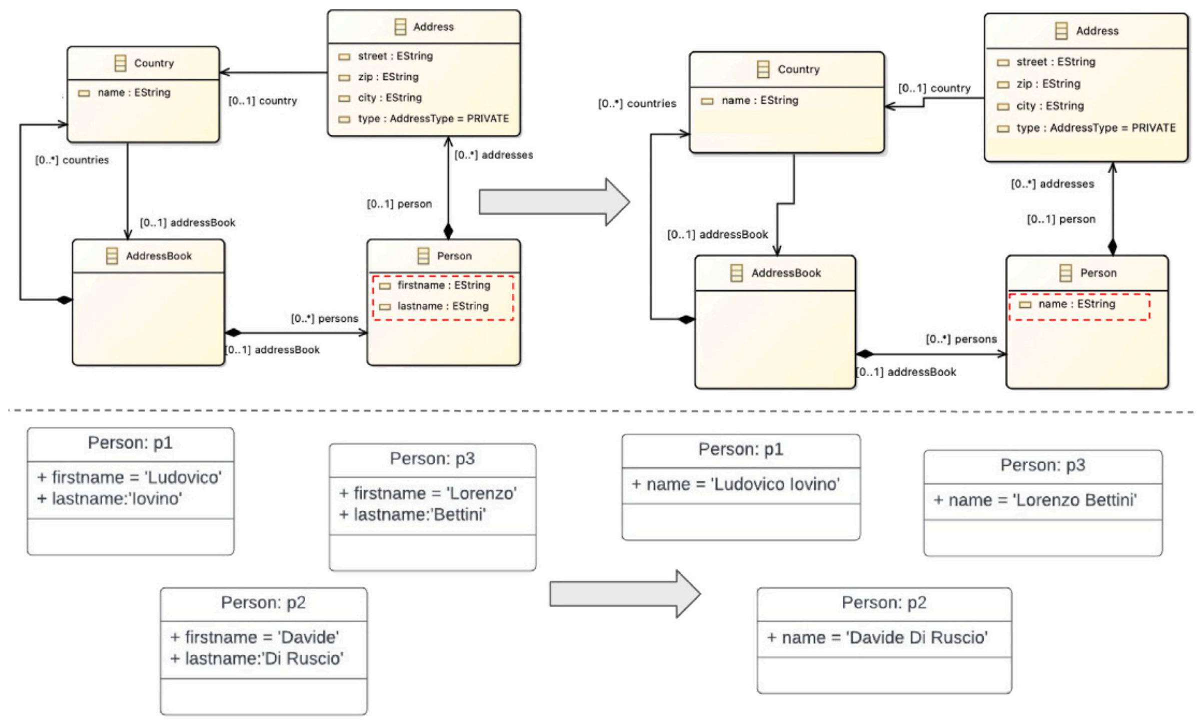


Fig. 5. Example of Merge attributes migration.

simple; the NAME attribute is set to the FIRSTNAME and LASTNAME values concatenation with a white space in the middle. This migration strategy is *domain-dependent* even if it can be applied by default to all the metamodels where the evolution pattern occurs (Mig_1).

In other cases, this pattern can be automated in different ways. For instance, if applied to a metaclass ACCOUNT, where first name and last name are merged into a USERNAME attribute, the migration strategy can be applied as a concatenation of firstname+"."+lastname, e.g., 'ludovico.iovino' (Mig_2), since white spaces are not allowed in accounts. Again, this pattern is *domain-dependent* even if the metamodels, including metaclasses representing accounts, are multiple. Other possible migration strategies could apply heuristics or, for instance, require the user to provide the separator. In the same metamodel in Fig. 5, if the merged attributes are STREET, ZIP, CITY to a COMPLETEADDRESS, the migration could require a separator, e.g., comma, to migrate the existing instances (Mig_3).

In the repository presented in Barriga et al. (2020), six metamodels reporting the metaclass PERSON are candidates for this possible migration resolution strategy. In particular, Mig_1 can be *domain-specific*, but it can also be used as a default strategy and then *domain-independent* as Mig_3, whereas Mig_2 seems to be domain-specific. The opposite evolution pattern of "merge attributes" is "split attributes", which can be managed as merge by identifying the separator and assigning the values to the split attribute's values. There are cases in which is not possible to deal with it without user intervention, for instance, in the case of multiple types of separators or undecidable resolutions. For example, in the case of strings containing multiple separators, an autonomous resolution would not be able to determine which separator(s) must be used to split the string into multiple attributes.

2.3.3. Extract metaclass

This metamodel evolution pattern is used when a metaclass violates the single responsibility principle, modeling unrelated content. In this case, a new metaclass can be extracted by moving all the attributes inside and associating a new reference. Examples of extraction of metaclass can be found in all Petri net examples, as for instance, in Cicchetti et al. (2008a), Taentzer et al. (2012).

An example of this evolution pattern is depicted in Fig. 6, where the metaclass PERSON contains multiple address-related attributes that can be moved to a new metaclass. In this case, this evolution pattern is a BRC since we can safely adopt a migration strategy that instantiates a new metaclass for every address, and the source person refers to the corresponding instance (Mig_1). Another variation of this evolution pattern can be used when instead of extracting an associated metaclass with optional multiplicity, the reference becomes required, i.e., instead of [0..1], we use [1]. This leads to possible migration strategies considering whether the attributes in the original metaclass are set or unset and derive the instantiation of the newly introduced superclass (Mig_2). Thus, this migration strategy can be considered *domain-independent* and can be used to deal with multiple metamodels.

2.3.4. Restrict/enlarge multiplicity

By taking as an example again the metamodel in Fig. 5, we can notice that a PERSON can have [0..*] associated ADDRESSES. Suppose the evolution considers restricting this cardinality to 1, for instance, as in Fig. 6 (evolved version on the right). In that case, the existing instances of PERSON are valid only in cases where the person has a single (or none) associated address, making the change NBC. Whereas, if the person has multiple addresses, the change is usually managed as BUC since the migration strategy cannot automatically drop instances of addresses without knowing which one is correct. In this case, if considered *domain-independent*, the change can be managed by user intervention and iteratively selecting the instances to drop (Mig_1). Another possibility can be to consider the first address as the most important and then maintain only the first one (Mig_2), considering the change as *domain-specific*. Other migration strategies could be designed for specific domains, for instance by specifically writing constraints to select the right ones. If the cardinality is enlarged, if the multiplicity was optional and becomes required, the change is BUC since at least one instance of the associated metaclass should be added to the model. In this case, heuristics can be used to define instances to make the model valid with the required task to make the instance consistent with the model at the refinement stage.

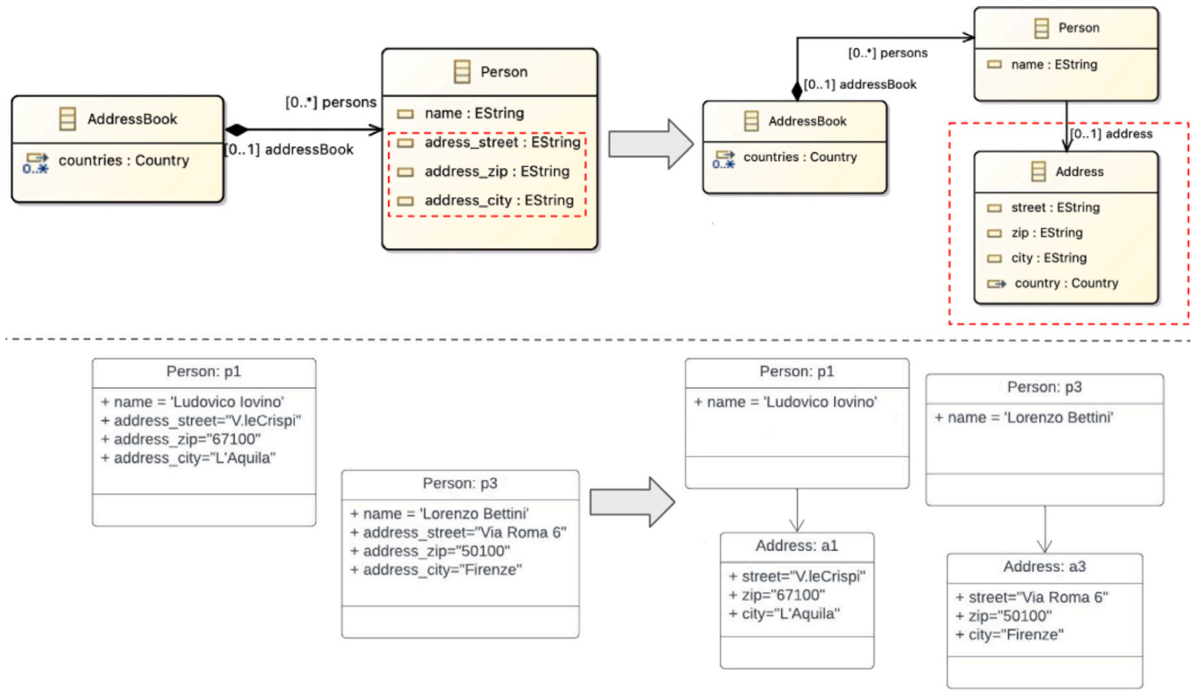


Fig. 6. Example of extract class migration.

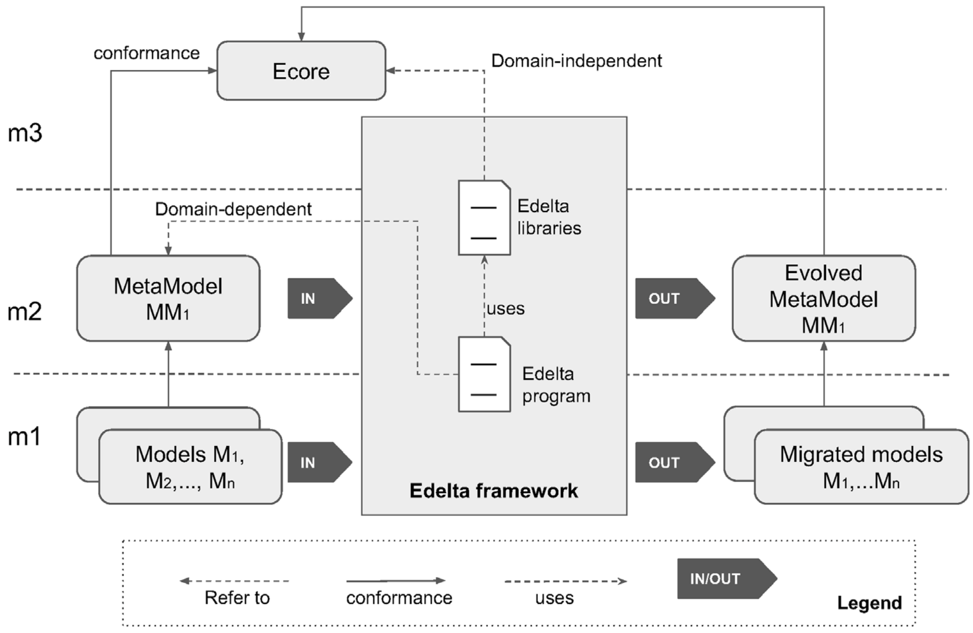


Fig. 7. Proposed Edelta extension to manage model migration.

3. Supporting reuse in model migration with Edelta

In this section, we propose a metamodel co-evolution approach that supports the re-use of migration patterns that can be defined as domain-specific, but also domain-independent. We implement our approach as an extension of the Edelta framework (Bettini et al., 2020b, 2022b), adding the new model management functionality that can be used to write migration strategies corresponding to metamodel evolution. The conceptual Edelta extension to manage model migration is reported in Fig. 7, and in the following sections, we give details about the implementation of the migration strategies in Edelta. In Fig. 7, “Edelta libraries” include both the Edelta Java runtime library (Section 3.1) and the migration library described in the rest of the paper.

In Section 3.1, we first recall how Edelta can be used to manage metamodel evolution. Then, in Section 3.2, we give details about the new application in model migration. In Section 3.3, we propose a classification to describe our implementations of migration patterns.

3.1. Evolving metamodels with Edelta

Edelta is an open-source project available at https://github.com/LorenzoBettini/edelta, and we also provide an Eclipse update site and a complete Eclipse distribution with Edelta installed. It is important to mention that we also provide a Maven plugin to compile Edelta programs headlessly, e.g., in a Continuous Integration server. Edelta

Listing 1: The shape of an Edelta program

```
metamodel "..."
metamodel "..."
...
use ... as ...
use ... as ...
...
def <name>(<... parameters ...>) : <
    returntype> {
  <body>
}
...
modifyEcore <name> epackage <EPackage name>
    {
  <body>
}
...
```

was originally conceived as a framework for refactorings and evolutions of EMF metamodels (Bettini et al., 2017). Still, it has also been applied with other intents, e.g., bad smell detection and resolution (Bettini et al., 2019), or for automatically detecting metamodel evolution in repositories (Bettini et al., 2020a). Since then, Edelta evolved with new features concerning the developer user experience (Bettini et al., 2020b) and additional static checks (Bettini et al., 2022b).

Edelta consists of a Java runtime library and a DSL to specify basic metamodel changes (e.g., additions and deletions) and complex reusable changes by aggregating existing library refactorings.

The Edelta runtime library is based on the Java API built on top of the standard EMF API. Still, it aims to provide a "fluent" style for operations specification and be more statically type-safe than the EMF one.

Concerning the Edelta DSL, a program consists of a few parts we report in pseudo-code in Listing 1. First, existing EMF metamodels are imported using the syntax metamodel followed by the EPackage's name. Then, existing Edelta libraries can be imported with the syntax use ... as ... to be used in the current program. Some reusable functions can be defined with a syntax starting with the keyword def (parameter declarations have the same syntax as in Java; the return type can be omitted and will be inferred by the compiler). Such functions can be used in the same program or imported into other Edelta programs. Finally, actual evolution operations on a specific imported EPackage can be specified with the syntax modifyEcore.

Concerning bodies of reusable functions and modifyEcore, the Edelta DSL provides a syntax similar to Java but removes much "syntactic" noise. For example, terminating semicolons are optional, and the parenthesis can be omitted when invoking a method without arguments. Edelta provides syntactic sugar for getters and setters: one can simply write o.name instead of o.getName() and o.name = ''...'' instead of o.setName(''...''). The Edelta DSL is statically typed, relying on type inference so that most types can be omitted in declarations. In particular, the type system of Edelta is completely compliant and interoperable with the Java type system so that an Edelta program can seamlessly use any existing Java code and Java libraries. Variable declarations in Edelta start with val or var, for final and non-final variables, respectively. In Edelta *lambda expressions* have the shape: [param1, param2, ...| body]. When a lambda is the last argument of a method call, it can be moved out of the parenthesis; for example, instead of writing m(..., [...]), one can write m(...)[...]. When a lambda is expected to have a single parameter, the parameter can be omitted and automatically available with the name it. The symbol it acts as an implicit receiver and can be omitted in method invocations, just like this. The DSL includes *extension methods*, a syntactic sugar mechanism to simulate adding new methods to existing types without modifying them. Instead of passing the first argument inside the parentheses of a method invocation, you call the method on the first argument (as if it were one of its methods). For example, if m(Entity) is an extension method, and e is of type Entity, you can write e.m() instead of m(e), as if m was a method defined in Entity.

For example, this is the implementation in Edelta of the "Introduction of subclasses" migration pattern shown in Section 2.3.1, dealing only with the metamodel (in the next subsection, we will also show the migration of the model). This reusable definition creates the classes with the given names as subclasses of the passed superClass, which will also be made abstract (note the syntactic sugar for setAbstract(true) and the use of addNewSubclass as an extension method):

```
def introduceSubclasses(EClass superClass,
    Collection names)
        : Collection<EClass> {
  superClass.abstract = true
  val subclasses = names.map[name |
      superClass.addNewSubclass(name)]
  return subclasses
}
```

The above function is part of our library EdeltaRefactorings, so we can apply this function to the example of Fig. 3, Section 2.3.1, in an Edelta program as follows (where we declare to use EdeltaRefactorings):

```
metamodel "statecharts"

use EdeltaRefactorings as refactorings

modifyEcore introduceNodeSubclasses epackage
     statecharts {
  val ePackage = it
  refactorings.introduceSubclasses(
    ecoreref(Node),
    #["InitialState", "FinalState", "State"]
  )
}
```

The special syntax for collections, #[e1, ..., en], allows the programmer to easily specify a list with initial contents. Note that Edelta provides a specific syntax to refer to Ecore elements in a statically typed way, ecoreref(...) so that programs refer directly to the model classes of an Ecore (like ecoreref(Node) in the listing above). Thus, this approach works even when the EMF Java model has not been generated. References to Ecore elements, such as packages, classes, data types, features, and enumerations, can be specified by their fully qualified name in an ecoreref expression using the standard dot notation or by their simple name if there are no ambiguities (possible ambiguities are checked by the compiler).

In the next sub-section, we will extend this example to the migration of models by using the extended version of Edelta presented in this paper.

The Edelta compiler translates Edelta programs into standard Java code, which relies on the Edelta runtime library. The generated Java code can be called from any Java program.

The Edelta DSL is embedded in an Eclipse-based IDE with all the typical IDE mechanisms, such as syntax highlighting, content assist, code navigation, quick fixes, incremental building, error reporting, and debugging.

We also provide an Eclipse wizard for creating an Edelta project with the required dependencies and an example Edelta program evolving an example Ecore file. The wizard also generates a Java file with a main method that calls the generated Java code. This Java main file is meant to be a starting point for the developer, who will customize it appropriately. We will see an example of such a Java main file in the examples of the next subsection.

The Edelta editor provides a "live" development environment for evolving metamodels, giving immediate feedback on the evolved version of the metamodels in the IDE. In fact, Edelta performs many static checks, also employing an interpreter that keeps track on the fly of the evolved metamodel, enforcing the correctness of the evolution right in the IDE based on the flow of the execution of the migration operations specified by the user. The interpretation is performed on an in-memory copy of the original metamodels, so modelers are free to experiment without affecting the original metamodels. These mechanisms allow for fast development cycles since the "live" preview is available even without saving the program in the editor.

To summarize, Edelta programs are much more compact than Java programs and much easier to read and understand based on a DSL.

For all the features of Edelta, its DSL, and its Eclipse-based IDE, we refer the interested reader to the Edelta website and to Bettini et al. (2020b).

In Fig. 7, we can place the previous status of Edelta applications in the $m2$ layer, where a metamodel can be programmatically evolved to produce an evolved version. In the next section, we describe how Edelta has been extended to express how the existing models are migrated when a specific metamodel evolution pattern is applied. In Fig. 7, this new functionality corresponds to the execution at $m1$ layer, but it includes all the modeling layers, from $m1$ to $m3$, and in the next section, we explain how.

3.2. Co-evolving models with Edelta

In the new version of Edelta presented in this paper, we enriched its Java API with mechanisms for co-evolving models and their metamodels. The input is then enriched with models that conform to the "evolving" metamodel that will be migrated in reaction to the specified evolution. Since the Edelta DSL is integrated with Java, no further extension to the Edelta DSL syntax was required.

Internally, the model migration in Edelta is implemented by a custom EMF EcoreUtil.Copier, which migrates EMF models together with the corresponding Ecore models. Such a mechanism takes care of the overall migration mechanism. The developer can participate in this evolution by specifying a few custom model migration rules, as we will explain in the rest of this section. Thus, the developer is relieved from the task of handling the model migration on a low level and can focus on some specific parts of the migration process in a declarative way.

In particular, the EcoreUtil.Copier itself implements a map associating original elements to copied elements; thus, in our custom implementation, we can always: (i) given a type element (respectively, an object) of the evolved metamodel (respectively, model), retrieve its original version (function getOriginal); (ii) given a type element (respectively, an object) of the original metamodel (respectively, model), retrieve its migrated version (function getMigrated); in particular, in this case, we can trigger the migration if a model element has not been migrated yet, making the model migration independent from the order of the objects in the model.

In both cases, for plain Java objects, not instances of EObject, these functions simply return the objects themselves. Moreover, these methods also work on collections. Finally, as prescribed by the contract of EcoreUtil.Copier, we first migrate all the contained elements and then all the references.

The new Edelta Java API for model migration consists of the facade class EdeltaModelMigrator. Thanks to the use of the custom copier described above and to the associations between original and evolved metamodels and models, our migrator automatically handles changes like renaming (since the associations work at the object identity levels, we do not need to lookup elements by their names) and removals (since in the evolved models the objects belonging to Ecore elements that have been removed are simply discarded). Besides that, the user can participate in the model migration, as shown in the next paragraphs.

In an Edelta program, from def and modifyEcore constructs, the method modelMigration can be called. This method takes as a parameter a lambda where the EdeltaModelMigrator methods can be used to specify custom model migration strategies for specific features and classes, which are meant to be related to the features and classes that have just been evolved in that program context.[12]

The EdeltaModelMigrator provides a few "rules" to define custom migration strategies. Rules are specified by calling specific methods of EdeltaModelMigrator. Such methods take two lambdas. The first one is a predicate, which has to determine when such a rule has to be applied. Our migrator calls this lambda with an Ecore element when it migrates the corresponding model element, e.g., an EClass, an EAttribute, etc. The second lambda is called when the predicate evaluates to true. The parameters of the second lambda depend on the rule that has been specified. For simpler rules, the lambda only takes a single element of the original metamodel or model and has to return the corresponding element of the evolved metamodel or model. For example, given a feature of the original metamodel, the lambda returns a feature of the evolved metamodel or an instance corresponding to that feature in the evolved model. For more complex rules, the lambda takes the feature of the original metamodel, the original model's object, and the evolved model's object. The latter has already been created by our migrator, and the lambda can further configure it, e.g., by setting its features' values as the developer sees fit.

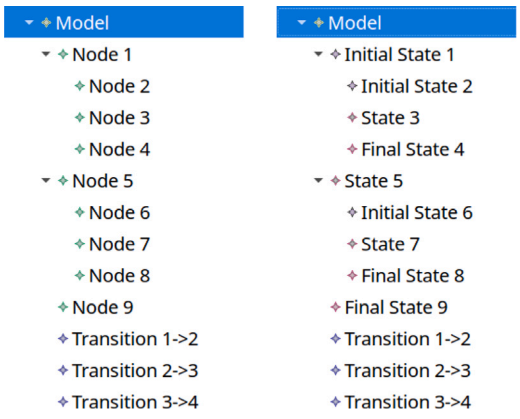
Concerning predicates for the rules, the EdeltaModelMigrator provides a few convenience methods for specifying such predicates: isRelated and wasRelated take an Ecore element (e.g., a feature or a type) belonging to the evolved metamodel and return a predicate to check whether the original element under migration is related to the argument. The latter does not check whether the element has been removed and thus allows the developer to provide migration strategies for elements that have been replaced by other elements in the evolved metamodel. Variants like wasRelatedToAtLeastOneOf(collection) check whether a single element is related to at least one element of the collection. This is useful in migrations that are related to several features. All these convenience methods are based on the associations maintained during the migration, as described above.

For the complete API provided by the EdeltaModelMigrator, we refer the interested reader to the Edelta website.

We can now extend the function introduceSubclasses of Section 3.1 to deal also with model migration. The function now needs an additional parameter, objectMigrator, which is applied for migrating objects that were originally instances of the superClass (as in Java, we use the functional interface java.util.function.UnaryOperator to represent a lambda expression that takes a single argument and returns a result of the same type as its argument):

```
def introduceSubclasses(EClass superClass,
        Collection names, UnaryOperator<
            EObject> objectMigrator)
        : Collection<EClass> {
  superClass.abstract = true
  val subclasses = names.map[name |
      superClass.addNewSubclass(name)]
  modelMigration[
    createInstanceRule(
      isRelatedTo(superClass),
      objectMigrator
    )
  ]
  return subclasses
}
```

[12] We suggest to rely on the implicit parameter it in such a lambda, so that the methods of EdeltaModelMigrator can be called without a receiver, as explained in Section 3.1. We use this methodology in the examples of the use of modelMigration in the rest of the paper.



- Model
  - Node 1
    - Node 2
    - Node 3
    - Node 4
  - Node 5
    - Node 6
    - Node 7
    - Node 8
  - Node 9
  - Transition 1->2
  - Transition 2->3
  - Transition 3->4

- Model
  - Initial State 1
    - Initial State 2
    - State 3
    - Final State 4
  - State 5
    - Initial State 6
    - State 7
    - Final State 8
  - Final State 9
  - Transition 1->2
  - Transition 2->3
  - Transition 3->4

Fig. 8. Mig_3: Example of migrated model of Fig. 3.

In this example, we use the rule createInstanceRule, which, besides the predicate on EClass, takes a lambda that is responsible for instantiating in the migrated model an object corresponding to the matched class in the original metamodel. The lambda takes as an argument the object of the old model, which can be used to properly initialize the object of the evolved model that the lambda has to instantiate and return. In this function, such a lambda (objectMigrator) has to be passed as an argument to this definition.

An application of this extended function to the example of Fig. 3, Section 2.3.1, is shown in the next listing. The first two arguments are the same as in Section 3.1. For the additional argument, we specify programmatically the migration strategy for objects that were of type Node in the original model. This strategy simply uses the position of the objects of the old model in the containing list to determine the concrete subclass in the evolved model (getValueAsList is one of our utility functions to safely convert values of a feature into a collection).

```
metamodel "statecharts"

use EdeltaRefactorings as refactorings

modifyEcore introduceNodeSubclasses epackage
     statecharts {
  val ePackage = it
  refactorings.introduceSubclasses(
    ecoreref(Node),
    #["InitialState", "FinalState", "State
       "],
    [oldObj |
      val nodes = oldObj.eContainer.
         getValueAsList(oldObj.
         eContainingFeature)
      if (nodes.head == oldObj)
        return createInstance(ePackage.
           getEClass("InitialState"))
      else if (nodes.last == oldObj)
        return createInstance(ePackage.
           getEClass("FinalState"))
      return createInstance(ePackage.
         getEClass("State"))
    ]
  )
}
```

If we apply this migration strategy to the model of Fig. 8-left, we get as an evolved model the one in Fig. 8-right.

As mentioned in Section 3.1, the Java code generated by the Edelta compiler is meant to be executed from a standard Java program (our project wizard generates an initial skeleton of such a Java main file). The typical steps are: create an instance of our EdeltaEngine passing an instance of the generated Java code corresponding to the Edelta program, loading the Ecore files and the model XMI files to be migrated, and call the execute method. Then, the evolved Ecore and model files can be saved into the specified output directory. For the example we have just shown, this is the Java main file, in which the metamodel and model to be migrated are passed at lines 6 and 7 and then executed by the Edelta engine at line 9:

```
 1 public class StateChartsExampleMain {
 2   public static void main(String[] args)
         throws Exception {
 3     // create the engine specifying the
           generated Java class
 4     EdeltaEngine engine = new EdeltaEngine
           (StateChartsExample::new);
 5     // Make sure you load all the used
           Ecores (Ecore.ecore is always
           loaded)
 6     engine.loadEcoreFile("model/
           StateCharts.ecore");
 7     engine.loadModelFile("model/
           StateChartsModel.xmi");
 8     // Execute the actual transformations
           defined in the DSL
 9     engine.execute();
10     // Save the modified Ecores and models
            into a new path
11     engine.save("modified");
12   }
13 }
```

3.3. Classification of Edelta migration patterns

All the implementations of migration patterns in Edelta are meant to be reusable across different metamodels. They are also meant to be automatic. We classify them according to the following definitions.

An **autonomous** migration pattern is a migration pattern that requires only values to be executed. Such a pattern can be implemented by a function whose arguments are meant to be fully evaluated before the invocation.

Thus, the arguments for such functions are meant to be Ecore elements, strings, integers, collections, and, in general, Java objects. Once such a function is called, it will automatically perform the migration.

A **collaborative** migration pattern is a migration pattern that, besides values, also requires some expressions that will be evaluated during the execution. Such a pattern can be implemented by a function that also accepts code blocks as arguments, meant to be used several times within the function when the function implementation sees fit.

Thus, some arguments for such functions are lambda expressions. Once such a function is called, it will still automatically perform the migration, but it will call the passed lambdas when some migration strategies cannot be implemented automatically and are delegated to the code blocks passed by the developer.

An **interactive** migration pattern is like an autonomous one, but it collaborates with the developer by using an interactive session.

Thus, such functions do not require code blocks and still automatically perform the migration. However, when some migration strategies cannot be implemented automatically, they prompt the user to decide how to migrate some model objects. The interactive session is any kind of user interface, e.g., a command line prompt or a GUI.

The patterns implemented in Edelta always perform checks to make sure that the arguments, excluding code blocks, satisfy some validity constraints (pre-conditions) for the specific metamodel evolution. In case of failed pre-conditions, the whole evolution process is interrupted with error reporting, thus ensuring we never produce invalid evolved metamodels/models. From the developer's point of view, when one of their code blocks is called or when they are prompted for an answer, they can assume that the arguments of the code block or the questions refer only to elements that need a proper migration because they are affected by that specific metamodel evolution pattern.

We will use this classification in the rest of the paper to describe our implementations of migration patterns. Although this classification is transversal to the ones introduced in Section 2 and specific to our Edelta approach, we can implement the non-breaking and breaking resolvable changes as autonomous: for the former, only the metamodel evolution must be specified; for the latter, the migration of the model can be derived by considering the evolution of the metamodel. The breaking unresolvable changes can be implemented both as collaborative and interactive patterns. However, if a breaking unresolvable change can be faced by using heuristics, then we can also implement such a change as an autonomous migration pattern.

4. Running examples

To show the application of model migration in Edelta, we show some Edelta functions that implement a few migration patterns from the examples of Section 2. The presented implemented functions are a subset of the refactoring library part of the Edelta distribution.

Please remember that all the implemented migration patterns described in our library are meant to be *domain–independent*. However, the developer may sometimes need to refer to domain-specific concepts in the migration specification. When an Edelta function cannot safely implement the model migration strategy in a completely autonomous way, an additional parameter in the shape of a lambda expression has to be passed to the function. The function will use such a lambda appropriately when the function needs to migrate the elements affected by the metamodel evolution. We also provide functions that prompt the user on the console while migrating metamodels and models instead of requiring a lambda. In such cases, some choices are shown to the user.

Thus, we divide our implementations according to the classification of Section 3.3.

4.1. Collaborative migration patterns

We have already shown an example of an implementation of a collaborative migration pattern, introduceSubclasses, in Section 3.2.

We now show the function of the Edelta library implementing the mergeAttributes migration strategy (Section 2.3.2). This function relies on the more general one, mergeFeatures (mergeReferences is similar):

```
def mergeAttributes(String newAttributeName,
     Collection<EAttribute> attributes,
    Function<Collection<Object>, Object>
        valueMerger) : EAttribute {
  return mergeFeatures(newAttributeName,
      attributes, valueMerger, null)
}

def mergeReferences(String newReferenceName,
     Collection<EReference> references,
    Function<Collection<EObject>, EObject>
        valueMerger, Runnable postCopy)
        : EReference {
  return mergeFeatures(newReferenceName,
      references, valueMerger, postCopy)
}

def <T extends EStructuralFeature, V>
    mergeFeatures(String newFeatureName,
        Collection<T> features, Function<
            Collection<V>, V> valueMerger,
        Runnable postCopy) : T {
  checkNoDifferences(
    features,
    new EdeltaFeatureDifferenceFinder().
        ignoringName,
    "The two features cannot be merged"
  )
  val firstFeature = features.head
  val owner = firstFeature.EContainingClass
  val mergedFeature = firstFeature.copyToAs(
      owner, newFeatureName)
  removeAllElements(features)
  modelMigration[
    copyRule(
      wasRelatedTo(firstFeature),
      [_, oldObj, newObj |
        var originalFeatures = features.
            stream()
            .map[a | getOriginal(a)]
        var oldValues = originalFeatures
            .map[f | oldObj.eGet(f)]
            .collect(Collectors.toList())
        var merged = valueMerger.apply(
            getMigrated(oldValues) as
            Collection<V>)
        newObj.eSet(mergedFeature, merged)
      ],
      postCopy
    )
  ]
  return mergedFeature
}
```

The function mergeFeatures performs the actual merging of features and the corresponding model migration after checking the constraint that the features have the same type, cardinality, and other properties (including that they belong to the same class). If the constraint does not hold, the function stops with a failure, describing the differences in the differing features and a message. After this check, since we can safely assume that all the features are compliant, we take the first one and copy that into the containing classes; all the passed features are then removed from the metamodel. We also note that the functions in Edelta can be made generic so that static type checking can be effectively performed.

In this example, we use copyRule, which, besides the predicate, takes a lambda with three parameters: the feature of the original metamodel (in this example, it is not used, so we use _ for the parameter name), the EObject of the original model (whose class has that feature) and the EObject of the model being migrated (which has already been created by the model migrator). The lambda is then responsible for setting the value (or values, depending on the context) in the object of the evolving model that is somehow related to that feature of the original metamodel. Remember that by default, our model migration strategy automatically discards the values (respectively, objects) related to features (respectively, types) that have been removed in the evolved metamodel. In this case, all the features of the original metamodel have been removed by this function in the evolved metamodel. Thus, we do not have to discard the corresponding values explicitly in the evolved model. Instead, we specify a rule that is applied when we encounter the feature of the original metamodel corresponding to the first feature passed to the function definition (we consider the first one in the predicate wasRelatedTo(firstFeature)). The lambda in the rule then simply merges the original values (relying on getOriginal) into a single one, using the merging function passed to this function.

The optional additional Runnable postCopy, if passed, will be executed after the model migration of the rule has been applied, that is, in this case, after the values of the original model have been merged into a single value of the evolved model. This is useful for merging references that refer to shared objects (i.e., not contained), for example, for performing some garbage collection of objects that are not referred to anymore in the evolved model after the merging.

Thanks to the architecture of our model migration implementation, the mergedFeature in the lambda of the model migration rule is exactly the same object as the new feature introduced in the evolved metamodel. Since newObj, created by our model migrator and passed as an argument to the lambda, is an object of the evolved metamodel, the EMF reflective operation newObj.eSet(mergedFeature, merged) will not fail at run-time: the feature mergedFeature belongs to the EClass of newObj.

The more specific functions mergeAttributes and mergeReferences only introduce, through their parameters, a specialization of the type arguments, allowing the compiler to perform a static check. This way, passing a collection with attributes and features is ruled out statically by the compiler. Similarly, a collection of attributes cannot be passed to mergeReferences.

An application of mergeAttributes to the model of Fig. 5 can be as follows:

```
metamodel "addressbook"

use EdeltaRefactorings as refactorings

modifyEcore mergeName epackage addressbook {
  refactorings.mergeAttributes(
    "name",
    #[ecoreref(firstname), ecoreref(lastname)],
    [oldValues | oldValues.filterNull.map[toString].join(" ")]
  )
}
```

In this case, the merging function that we pass to mergeAttributes consists of a simple joining with a space character.

4.2. Autonomous migration patterns

We now show the implementation of "Enum To Subclasses" (Section 2.3.1) as a particular case of "Introduce Subclasses" we have shown in Section 3.2. The presented listing is intentionally simpler than the actual implementation: we removed some checks and possible error reporting, e.g., if the attribute's type is not an enumeration.

```
def enumToSubclasses(EAttribute attr) : Collection<EClass> {
  val type = attr.EAttributeType as EEnum
  val owner = attr.EContainingClass
  val createdSubclasses = introduceSubclasses(owner,
    type.ELiterals.map[toString.toLowerCase.toFirstUpper],
    [oldObj |
      val literalValue =
        oldObj.getValueFromFeatureName(attr.name).toString()
      val correspondingSubclass =
        owner.findSiblingByName(literalValue.toLowerCase.toFirstUpper)
      return createInstance(correspondingSubclass)
    ]
  )
  removeElement(type) // will also remove the attribute
  return createdSubclasses
}
```

In this implementation, we call introduceSubclasses (Section 4.1) by passing an object migrator lambda that creates instances of the introduced subclasses according to the original value of the type attribute. Note that this function also removes such an attribute since, in the new metamodel, it is useless.

An application of enumToSubclasses to the example of Fig. 4, Section 2.3.1, is shown in the next listing where we also rename the introduced subclasses according to Fig. 4. Note that thanks to the on-the-fly interpretation of the Edelta compiler (see Section 3.1), the introduced subclasses are immediately available in the rest of the program.

```
metamodel "statechart"

use EdeltaRefactorings as refactorings

modifyEcore introduceNodeSubclasses epackage statechart {
  refactorings.enumToSubclasses(ecoreref(type))
  //optional renaming of the newly created classes
  ecoreref(Normal).name = "State"
  ecoreref(Initial).name = "InitialState"
  ecoreref(Final).name = "FinalState"
}
```

If the initial model is as in Fig. 8-left of the previous section, where the type attributes' values are set appropriately to enum literals, the resulting evolved model is the same as in Fig. 8-right.

The implementation of "Extract Metaclass" described in Section 2.3.3 in Edelta is as follows:

```
def extractClass(String name, Collection<EStructuralFeature> features) {
  checkNoBidirectionalReferences(features,
    "Cannot extract bidirectional references")
  val owner = findSingleOwner(features)
  val extracted = owner.addNewEClassAsSibling(name)
  val reference = owner.addMandatoryReference(name.toFirstLower, extracted)
  makeContainmentBidirectional(reference)
  features.moveAllTo(extracted)
  modelMigration[
    copyRule(
      wasRelatedToAtLeastOneOf(features),
      [origFeature, origObj, migratedObj |
        var extractedObj = migratedObj.getOrSetEObject(reference,
          [extracted.createInstance])
        extractedObj.eSet(
          getMigrated(origFeature),
          getMigrated(origObj.eGet(origFeature))
        )
      ]
    )
  ]
  return reference
}
```

We first check a few constraints: we cannot extract a feature that is a bidirectional reference, and we must ensure that all the passed features belong to the same class (findSingleOwner). Then, we create a new class (in the package of the owner of the features, addNewEClassAsSibling), and we add to the owner a required containment reference to the new class, which is then made bidirectional (makeContainmentBidirectional). Finally, all the features are moved to the new extracted class. Concerning model migration, we apply the same rule to all the features of the original metamodel that are related to the (now moved) features of the evolved metamodel. We use again copyRule, explained in Section 4.1 for mergeFeatures. In this case, the lambda of the rule uses all the passed arguments. In particular, we create an instance of the extracted class (only the first time) and set the corresponding values. Note the use of getMigrated that we described before.

An application to the example of Section 2.3.3, Fig. 6 is as simple as this:

```
metamodel "addressbook"

use EdeltaRefactorings as extension
    refactorings

modifyEcore extractAddress epackage
    addressbook {
  refactorings.extractClass(
    "Address",
    #[ecoreref(street), ecoreref(zip),
        ecoreref(city)]
  )
}
```

Restricting and enlarging multiplicity (described in Section 2.3.4) are provided through the following functions in our Edelta library. We provide two convenience functions (changeToSingle and changeToMultiple) that are implemented in terms of the most generic one, changeUpperBound:

```
def changeToSingle(EStructuralFeature
    feature) {
  changeUpperBound(feature, 1)
}

def changeToMultiple(EStructuralFeature
    feature) {
  changeUpperBound(feature, -1)
}

def changeUpperBound(EStructuralFeature
    feature, int upperBound) {
  feature.upperBound = upperBound
  modelMigration[
    copyRule(
      isRelatedTo(feature),
      multiplicityAwareCopy(feature)
    )
  ]
}
```

In changeUpperBound we also implement a default strategy for model migration: excluding the extra elements, if any, according to the new upper bound. This is implemented by the model migration Java utility function of the Edelta library multiplicityAwareCopy that returns a lambda that considers at most n elements, where n is the upper bound of the migrated feature:

```
Procedure3<EStructuralFeature, EObject,
    EObject> multiplicityAwareCopy(
        EStructuralFeature feature) {
   return (EStructuralFeature oldFeature,
       EObject oldObj, EObject newObj) -> {
      if (oldObj.eIsSet(oldFeature))
         EdeltaEcoreUtil.setValueForFeature(
            newObj,
            feature,
            getMigrated(
               EdeltaEcoreUtil.
                   getValueForFeature(oldObj
                   , oldFeature,
                      feature.getUpperBound())
                        )
          );
   };
}
```

In this code snippet, we use our utility functions getValueForFeature and setValueForFeature, which allow the developer to treat the model's values uniformly as collections. The former returns a collection of, at most, the specified size (−1 means without limit), even in case the value corresponds to a single feature. In that case, the single value is wrapped in a collection if the value is set or in an empty collection otherwise. The latter implements the unwrapping of the collection in case of a single feature. This way, the developer does not have to check the cardinality of features and can then write simpler migration strategies than with standard EMF reflective API. Moreover, we use getMigrated to retrieve the objects in the evolved model corresponding to those in the original model. As said before, getMigrated also triggers the copy of such values if they have not been copied already.

4.3. Interactive migration patterns

In the next listing, we show an alternative to introduceSubclasses of Section 3.2, where no lambda for model migration is required. In this variant, we delegate to the introduceSubclasses we showed before by passing a lambda for model migration that prompts the user to provide the appropriate subclass while migrating each object of the original superclass. This relies on some utility functions we provide in Edelta: the attributes' values of the current object are shown on the console, together with its position inside its container (obtained through the utility function EdeltaEObjectHelper.positionInContainter). Then, the possible subclasses' names are shown, and the user must select one.

```
use EdeltaRefactorings as refactorings

def introduceSubclassesInteractive(EClass
    superClass, List<String> names)
        : Collection<EClass> {
  return refactorings.introduceSubclasses(
      superClass, names) [
    oldObj |
    val helper = new EdeltaEObjectHelper
    EdeltaPromptHelper.show("Migrating " +
        helper.represent(oldObj))
    EdeltaPromptHelper.show(helper.
        positionInContainter(oldObj))
    val choice = EdeltaPromptHelper.choice(
        names)
    return createInstance(superClass.
        EPackage.getEClass(choice))
  ]
}
```

If we apply this migration to the metamodel of Mig_3 defined in Section 2.3.1 and to the model of Fig. 8-left, we get this output on the console. As an example, we also show some inputs provided by the user (what the user inserts is the number after the "?"):

```
Migrating Node{name = 1}
1 / 3
 1 InitialState
 2 State
 3 FinalState
Choice? 1
Migrating Node{name = 2}
1 / 3
 1 InitialState
 2 State
 3 FinalState
```

```
Choice? 4
Not a valid choice: 4
Choice? 1
Migrating Node{name = 3}
2 / 3
 1 InitialState
 2 State
 3 FinalState
Choice? 2
...
```

In this case, the user uses the position of the node object being migrated to specify the subclass in the migrated model. If the choices are made consistently, the result is the same as Fig. 8-right.

We also show a possible implementation for merging string attributes (see mergeAttributes in Section 4.1) prompting the user to specify the separator character (note that for this migration pattern, if we want to implement an interactive version, we must be explicit on the type of the attribute):

```
metamodel "ecore"

use EdeltaRefactorings as refactorings

def mergeStringAttributes(String newAttributeName,
        Collection<EAttribute> attributes) : EAttribute {
  refactorings.checkType(attributes.head, ecoreref(EString)) // fails if not EString
  refactorings.mergeAttributes(
    newAttributeName,
    attributes,
    [oldValues|
      val stringValues = oldValues.filterNull.map[toString]
      if (stringValues.empty)
        return null
      EdeltaPromptHelper.show("Merging values: " + stringValues.join(", "))
      val sep = EdeltaPromptHelper.ask("Separator?")
      return stringValues.join(sep)
    ]
  )
}
```

Applying mergeStringAttributes for migrating the model of Fig. 5 can be as follows:

```
metamodel "addressbook"

use EdeltaRefactoringsWithPrompt as extension refactorings

modifyEcore mergeName epackage addressbook {
  refactorings.mergeStringAttributes(
    "name",
    #[ecoreref(firstname), ecoreref(lastname)]
  )
}
```

We show an example of console interaction based on the model of Fig. 5. In the example of console interaction, the space characters inserted by the user are represented as :

```
Merging values: Ludovico, Iovino
Separator?
Merging values: Davide, Di Ruscio
Separator?
Merging values: Lorenzo, Bettini
Separator?
```

Note that with this version, the user could also specify a different separator for each migrated object.

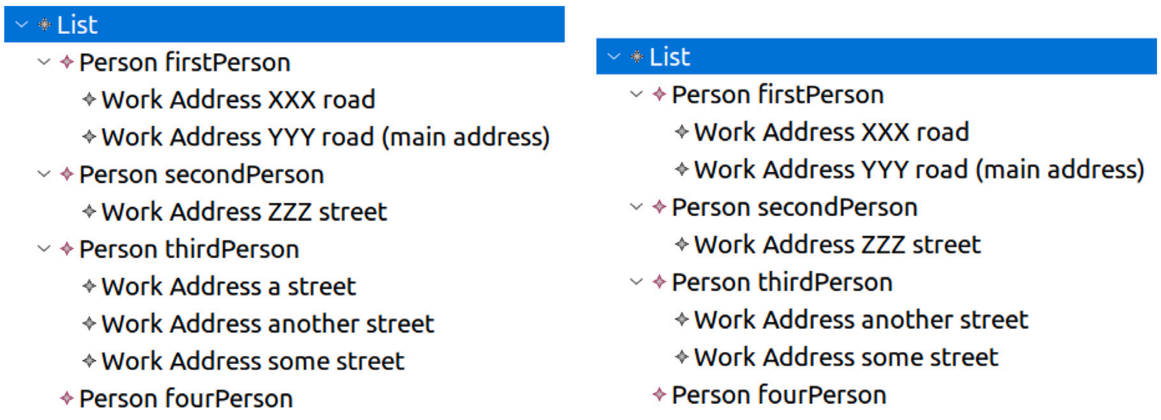
We also show an interactive version of changeUpperBound (Section 4.1) that, in case the elements to migrate are more than the migrated feature's new upper bound n, asks the user to select n elements (discarding all the remaining ones), by showing the possible choices:

```
def changeUpperBoundInteractive(
    EStructuralFeature feature, int upperBound) {
  feature.upperBound = upperBound
  modelMigration[
    copyRule(
      isRelatedTo(feature),
      [origFeature, origObj, migratedObj |
        var origValues = origObj.getValueForFeature(origFeature, -1)
        if (origValues.size <= upperBound) {
          migratedObj.setValueForFeature(feature, getMigrated(origValues))
          return
        }
        val helper = new EdeltaEObjectHelper
        EdeltaPromptHelper.show("Migrating " + helper.represent(origObj))
        val choices = origValues.map[helper.represent(it)].toList
        val newValues = new ArrayList(upperBound)
        for (var i = 1; i <= upperBound; i++) {
          EdeltaPromptHelper.show("Choice " + i + " of " + upperBound)
          val choice = EdeltaPromptHelper.choiceIndex(choices)
          val chosen = origValues.get(choice)
          newValues.add(getMigrated(chosen))
        }
        migratedObj.setValueForFeature(feature, newValues)
      ]
    )
  ]
}
```

In this code snippet, we also use getValueForFeature, setValueForFeature, and the console interaction functions described before.

For example, let us apply this latter function to the feature "workAddress" of the model in Fig. 9-left by reducing the upper bound to 2 (originally, the upper bound was -1). The migration pattern prompts the user only for the thirdPerson. If we imagine a console session like the following one, the resulting model will be as in Fig. 9-right:

```
Migrating PersonListForChangeUpperBound.Person{firstname
= thirdPerson}
Choice 1 of 2
 1 PersonList.WorkAddress{street = a street, houseNumber = 3}
 2 PersonList.WorkAddress{street = another street, houseNumber = 3}
 3 PersonList.WorkAddress{street = some street, houseNumber = 3}
Choice? 2
Choice 2 of 2
 1 PersonList.WorkAddress{street = a street, houseNumber = 3}
 2 PersonList.WorkAddress{street = another street, houseNumber = 3}
 3 PersonList.WorkAddress{street = some street, houseNumber = 3}
Choice? 3
```


List
Person firstPerson
Work Address XXX road
Work Address YYY road (main address)
Person secondPerson
Work Address ZZZ street
Person thirdPerson
Work Address a street
Work Address another street
Work Address some street
Person fourPerson

List
Person firstPerson
Work Address XXX road
Work Address YYY road (main address)
Person secondPerson
Work Address ZZZ street
Person thirdPerson
Work Address another street
Work Address some street
Person fourPerson

Fig. 9. Example of migrated model with changeUpperBoundInteractive.

5. Evaluation

The evaluation aims to confirm Hypothesis 1 defined in Section 2, i.e., the migration strategy may be re-used across metamodels. To test the hypothesis, we defined the following research questions:

> **RQ3:** Can the proposed Edelta extension implement the co-evolution patterns extracted from the literature?
> **RQ4:** Can the proposed Edelta extension support cross-domain model migration?

5.1. Validation of the migration patterns implemented in the Edelta library (RQ3)

To answer RQ3, we validated our implementation of metamodel and model migrations by writing automated tests with JUnit. First, we implemented the new model migration parts, i.e., the EdeltaModelMigrator and the custom EcoreUtil.Copier described in Section 3.2, using the Test-Driven Development methodology (Beck, 2003). Such a procedure allowed us to gain confidence in the correctness of the general model migrations performed together with the corresponding metamodel migrations. It also permitted us to verify that basic changes like renaming and removals are working correctly. For our JUnit tests, we created several ad-hoc metamodels and models. These metamodels and models have no semantic meaning: they only represent input data specific to the migration scenario under test. They are also meant to cover our code fully, including exceptional and corner cases. We wrote several JUnit assertions that check that the migrated metamodels and models are as expected, e.g., a renamed feature in the metamodel induces consistent renaming in type references in the corresponding migrated models. In that respect, we manually verified that the migrated metamodels and models were as expected on the first test run. Then, we wrote the corresponding JUnit assertions, always to be checked as part of our Continuous Integration process to catch possible future regressions and bugs. Note that during our JUnit tests, the standard EMF validators are also used to check that the migrated metamodels and models are still valid. Then, we applied the same methodology to implement the migration patterns described in the paper. We implemented a few JUnit tests for each migration pattern to cover its implementation, including corner cases. Again, we wrote ad-hoc metamodels and models to recreate the meaningful input data for testing the implementation of the migration pattern.

As shown in the examples of this paper, our implemented migration functions are compact because they rely on our utility function library for inspecting, navigating, and checking EMF models. These utility functions had already been tested in isolation, so their correctness is assumed and must not be tested again when implementing the migration patterns. It is worth noting that in this development process, apart from the implemented code, the lengthy and delicate part is the first manual check that the migrated models are as expected. After that, once the JUnit tests assertions have been written, running the whole test suite for the migration functions only takes a few seconds.

Finally, we applied our migration functions to the example metamodels and models once our migration functions were tested. We applied the same testing methodology in this case as well. Of course, in such cases, while the models are still ad-hoc, the corresponding metamodels are the ones of the examples.

We have relied on automated tests since the beginning of the implementation of Edelta and its DSL. For each commit on the Edelta GitHub repository, which includes all our migration functions, an automatic build process is started on GitHub Actions. This automated continuous integration process runs all the tests (unit and integration tests, including end-to-end tests against the Eclipse distribution we make available to our users). We use all the virtual environments provided by GitHub Actions, e.g., Linux, Windows, and macOS. Moreover, our build on GitHub Actions also keeps track of code quality metrics, like code coverage (using the cloud service Coveralls[13]) and static analysis using SonarCloud[14] (a cloud service based on the well-known code quality analysis tool SonarQube).

5.2. Applicability of the migration patterns implemented in Edelta (RQ4)

To answer RQ4, we formulated Table 2 starting from Table 1. In Table 2, we report the metamodel example elicited from the cited paper in the first column. We used a bullet icon to populate the cells of Table 2. A bullet of the shape ● means an Edelta autonomous migration pattern (Section 4.2) is available in the provided library and then can be used in that example. A bullet of the shape ◐ means an Edelta collaborative or interactive migration pattern (Sections 4.1 and 4.3) is available in the provided library and can be used in that example. The indicated migration strategy for each example corresponds to the one reported in the cited paper. We highlight that multiple versions and variations of Petrinet metamodels are present in the literature, and, in particular, the second and third lines report multiple evolution scenarios contained in the same papers. Similarly, this occurs for UML activity diagrams. The last row represents a summary line reporting the number of groups of metamodels affected by the migration pattern.

"Extract Metaclass" occurs in different metamodels, i.e., Petrinet and SQL. Moreover, multiple Petrinet metamodels and multiple migration patterns are reported. In all the Petrinet examples, the application of the Edelta implementation of "Extract Metaclass" can be re-applied as autonomous patterns. The column of "Introduction of Subclasses" contains both ● and ◐ bullets. As seen in Sections 2.3.1 and 4.1,

[13] https://coveralls.io/
[14] https://www.sonarsource.com/products/sonarcloud/

Table 2
Evaluation Results: Reuse of co-evolution patterns in different metamodel examples.
● = Edelta Autonomous migration pattern ◐ = Edelta collaborative or interactive pattern.

Examples	Co-evolution patterns																			
	Extract metaclass	Extract super metaclass	Merge attribute	Split attribute	Add attribute	Delete attribute	Rename metaclass	Introduction of subclasses	Collapse Hierarchy	Inline class	Move Reference	Rename metaclass	Enlarge/Restrict multiplicity	Add reference	Merge metaclass	Rename attribute	Rename reference	Delete reference	Split reference	Move/Push down Attribute
Petrinet (Rose et al., 2010a,b)	●	●			●															
Petrinet (MM0–>MM2) (Cicchetti et al., 2008a)	●												●							
		●			●		●													
Petrinet (E0–>E2) (Taentzer et al., 2012)	●	●			●															
	●																			
Petrinet (Di Ruscio et al., 2012; Wagelaar et al., 2012)	●	●			●		●													
Petrinet (Krause et al., 2013)	●				●			◐												●
Petrinet (Anguel et al., 2014)	●																			
StateChart (Kessentini et al., 2016)								◐												
ExamXML (García et al., 2012)				●		●		◐												
Company (Rutle et al., 2020)			●				●	●	●	●	●									
excerpt SQL2003 (Garcés et al., 2014)	●	●			●	●						●								
Component and communication (Demuth et al., 2016)													●	◐						
Activity Diagram (Taentzer et al., 2013)								●					◐	◐						
UML Activity Diagram (Rose et al., 2014a)							●	●	●	●					●		●			
Project Management (Kessentini and Alizadeh, 2020)								◐			●									●
Workplace (Di Ruscio et al., 2016)								◐	●				◐							●
Component and connector (Rose et al., 2014b)								◐			●									
UML Class Diagram (Rose et al., 2014b)						●	●							◐	●	●	●	●	◐	
Summary	2	2	1	1	2	3	4	9	3	2	3	1	4	3	2	1	2	1	1	3

that co-evolution cannot be autonomous: the developer must decide how the instances of a single class in the original model have to be re-typed according to the new subclasses introduced in the evolved metamodel. However, in some examples, we need the specialization where an enumeration is converted into subclasses according to the original enumeration literal (Section 2.3.1). As shown in Section 4.2, the implementation of such a special case, enumToSubclasses, is an autonomous migration pattern. Concerning the change of multiplicity, we have shown in Section 4.2 that we provide an autonomous migration pattern. Nevertheless, the safe automatic migration strategy of changing the size of the values of the corresponding evolved collection might not fit some special cases. In Section 4.3, we provide an interactive version that the developer can use to decide which elements must be kept in the evolved collection. Also, the pattern "Enlarge/Restrict multiplicity" is marked as ● and ◐ since the pattern restricting the cardinality of a reference may lead to multiple resolution strategies that can involve autonomous in case of heuristics application or collaborative and interactive in case user intervention is needed. For instance, when reducing the cardinality in a metamodel reference from 1..* to n..m, if the existing referenced instances are more than m, an autonomous pattern would not be able to decide which instances to unlink or remove.

Table 2 confirms what we expressed in RQ4 since for all the examples belonging to different domains, we can use the same library of co-evolution patterns EdeltaRefactorings. We need to import the library with a use clause in the examples of Section 4 and Appendix. Such a library is part of the standard Edelta distribution. The source file EdeltaRefactorings.edelta can be browsed online at https://github.com/LorenzoBettini/edelta.[15]

6. Discussion

We can notice, from Table 2, that only a few patterns in the literature have been applied to single domains, e.g., *Split or Merge attributes*. While all the others confirm that the strategies can be shared across multiple domains. The extended version of the Edelta framework allowed us to propose a library of implemented migration patterns that handle the co-evolution of metamodels and models. If a metamodel evolution pattern is suitable to automatically handle the corresponding model evolution, our library functions also autonomously migrate the models' elements corresponding to the evolved metamodel elements.

[15] This library is still under development and is updated frequently.

By design choice, we implement an autonomous co-evolution of models only when it is safe to do so in a completely automatic way. Thus, we never apply automatic model evolution strategies that could result in invalid evolved models. We remind that the first aim of the co-evolution activity is to restore the model's validity, and the second is to obtain qualitative migrated models (Hebig et al., 2013).

Despite ensuring valid evolved models corresponding to valid evolved metamodels, some automatic strategies might not suit some involved semantic properties of the model that cannot be expressed with static constraints. For example, in changeUpperBound presented in Section 4.2, it is safe to "truncate" a collection of values after decreasing the upper bound in the metamodel. However, the modeler, and only the modeler, of that specific domain, might know the right collection values that must be kept. For this reason, as shown in Section 4.3, we also provide alternative interactive versions of the migration patterns allowing the modeler to decide which values to keep.

Some migration patterns cannot safely implement an autonomous model evolution strategy. In such cases, our refactoring functions require some collaboration from the developers: additional code block arguments. The developer responsibility, who knows that specific domain, is to provide such code blocks. We want to stress that such a code block is the minimal requirement that our functions need. Even in such cases where the model migration cannot be fully autonomous, Edelta takes over most of the model migration, removing most responsibilities from the developer. The code blocks provided by the developer will automatically be used by the Edelta migration function at the right moment and only on the right elements. Thus, our implementations always automatically know which model elements must be migrated. Sometimes, they cannot provide an automatic migration strategy for such elements. The developer's code blocks are ensured to be used only when needed and with the right elements. So, the developer's effort is kept to a minimum.

Note that, despite our automatic management of elements to be migrated, when the developer needs to perform some complex migration of such elements, our library allows the developer to do that, still minimizing the effort. An example of such a case is the last example of Appendix. In that case, our function splitClass is called for the common and reusable strategies. The developer can provide a code block mapping a single object of the original model into possibly two objects in the evolved model.

It is also important to stress that all our library functions perform several constraint checks before applying the evolution of the metamodel (and, thus, of the model). We have shown a few examples of constraint checks in Section 4. Thus, using our library means reusing the evolution operations and all the constraint checks. Such evolutions and checks have been thoroughly tested in Edelta. If a developer wanted to re-implement a common migration pattern from scratch, such checks should also be re-implemented from scratch.

To summarize, we believe using our library provides safety guarantees and reusable operations. Using our library cannot be harder or more time-consuming than re-implementing the co-evolution operations from scratch.

As described in Section 3.2, our metamodel and model migration process heavily relies on a custom EcoreUtil.Copier. Thus, as implied by EcoreUtil.Copier, during the migration, there will always be two copies in memory of each metamodel and model involved in the migration (one copy for the original one and one copy for the migrated one). Since our approach is meant to be generic and reusable, performing a migration in place without keeping two copies of the element being migrated would not be feasible. Moreover, using a EcoreUtil.Copier is a standard procedure for dealing with versions of EMF models (see, e.g., "EMF Diff/Merge"). Our approach does not add further overhead concerning memory. For checking, visiting, and traversing EMF models, we rely on the EMF API. In particular, for dealing with cross-references, we rely on the EMF utility classes of EcoreUtil (e.g., EcoreUtil.CrossReferencer and EcoreUtil.UsageCrossReferencer). These are already optimized to deal with large sets of models very efficiently. Note that we require inter-dependent models to be loaded in the same EMF ResourceSet during the migration. Again, this is required to let EMF correctly deal with cross-references. However, different clusters of inter-dependent models can be migrated at different times, avoiding the use of too much memory. Thus, we believe our approach can be used efficiently and scalable in every context where EMF can be used efficiently, even with large models. In that respect, we plan to perform experiments and benchmarks with very large inter-dependent models in the future.

7. Threats to validity

Threats to internal validity are related to factors or inaccurate settings that can influence the evaluation outcomes. The implemented examples are based on the case studies selected from the literature. The current implementation of the Edelta library supporting co-evolution is still under development and will be continuously updated to support more patterns, even though most of the most used patterns are already implemented. Another threat to internal validity regards the migration strategies classified as BUC, in which Edelta proposes a prompt to the modeler to provide additional information needed by the migration program. It is clear that in large models, this mechanism can be really time-consuming and less user-friendly than using an ad-hoc migration strategy. We tried to mitigate this point by using models contained in the projects, when available, or reproducing the models reported in the paper proposing the case study. However, we know that a more extended experiment involving users is required, which is part of the future plans. Moreover, we are already working on a GUI replacing the prompt and the integration of AI in these specific cases.

For these reasons, we do not advertise the use of refactorings with prompts. These must be considered simple and fast prototyping mechanisms to experiment with possible migration strategies that cannot be fully automated. We still foster the use of our functions that take a lambda (code block) as an argument for those refactorings that cannot be fully automated. This also allows for easier automated testing mechanisms and can scale to large models, which the prompting mechanism cannot easily do.

Threats to external validity are related to the validity of the results outside the considered setting. It is clear that this work is built in the context of EMF since Edelta is based on EMF APIs, so we think that this point is less problematic than others. Moreover, the Edelta approach can be implemented in other languages outside the EMF technical space.

8. Related work

In Paige et al. (2016), three categories of co-evolution approaches have been identified: manual approaches, inference approaches, and operator approaches. Note that the breaking unresolvable changes cannot be fully automated. They require the metamodel's user intervention since she knows how to apply the correct migration actions to restore the model conformance.

Manual approaches usually offer dedicated languages to specify migration strategies, which migrate models to an updated metamodel. An example in this category is Ecore2Ecore (Paternostro and Hussey, 2006), an EMF-specific tool specifying inter-relationships between original and evolved metamodels. The tool automatically generates a partial migration strategy. The approach does not seem to support complex evolution patterns, but only very trivial ones. Another manual approach is presented in Taentzer et al. (2013) where co-evolution rules are specified using graph transformations. Migration rules are specified graphically using a visual modeling language based on the changed metamodel and operators to manage the migration strategies.

Wimmer et al. (2010) proposed another manual approach in which the model co-evolution problem was solved using three steps. The approach automatically merges the source (MM_1) and target metamodels

(MM_2) in the first step. Then in the second step, the user can manually define *inplace transformations* to specify the co-evolution rules. These transformations allow adding the evolved model elements to the original model. The inplace transformation can be specified using any transformation language, and the authors used the ATL language. In the end, a model-to-model transformation is applied to remove all original model elements. The output model conforms to the MM_2 metamodel and is the co-evolved model. The approach has two advantages: on the one hand, the user does not need to specify evolution rules for unchanged elements; on the other hand, the user can use a known transformation language to define co-evolution rules.

Inference approaches are based on comparison or differencing of the original and updated metamodel and automated generation of evolutionary strategies for models. Such approaches as, for instance, Cicchetti et al. (2008a) and AML (Garcés et al., 2009), use metamodel comparison algorithms to detect metamodel changes and infer the migration strategies to be applied.

Operator approaches are pattern-based and typically provide a dedicated language to express model migration strategies.

COPE (Herrmannsdoerfer et al., 2009) provides a set of primitives for metamodel adaptation and model migration. For example, like in our approach, these primitives allow the creation, renaming, and deleting of elements. The user can use these primitives to define custom migration strategies in Groovy[16] (COPE also provides a Groovy editor with basic syntax highlighting). COPE also offers some reusable migration patterns, like the ones we presented in this paper, which can be used when facing recurring migration patterns. Migration patterns in COPE are applied directly in the EMF Ecore editor, extended by COPE. COPE records metamodel evolution changes in a "history" while the modeler applies them in the extended Ecore editor. From the history, the user can generate a migrator that will be used to perform the actual coupled metamodel/model evolution.

Our approach aims at the same goals of COPE: "combine the reuse of recurring transformations with the expressiveness to cater for complex transformations" (Herrmannsdoerfer et al., 2009). Although the approaches are similar and have the same intent, we believe Edelta and its DSL are less error-prone and easier to use, as detailed in the following. First of all, our DSL is statically typed (differently from Groovy), though it still has the compact shape of dynamically typed languages since it relies on type inference, so most types can be omitted in declarations. In particular, the type system of Edelta is completely compliant and interoperable with the Java type system so that an Edelta program can seamlessly use any existing Java code and Java libraries (Section 3.1). Thus, our DSL allows many programming errors to be caught early by the compiler, not later, while running the program (in this case, the migrator). Moreover, as happens for statically typed languages, our editor is enriched with more advanced IDE support. As described in Section 3.1, the Edelta DSL editor does not only provide syntax highlighting: it provides all the typical IDE mechanisms, such as content assist, code navigation, quick fixes, incremental building, and error reporting. In particular, it allows debugging the original Edelta DSL specification when executing the generated Java code while providing the typical Eclipse debugging views like breakpoints (possibly condition-based), variables, etc. Moreover, thanks to the on-the-fly interpreter (Section 3.1) used by the Edelta DSL compiler, the developer always has immediate feedback on the evolved version of the metamodels in the IDE: the Eclipse Outline View shows the modified parts of the metamodel and the sections in the DSL specification where that part of the metamodel has been modified (more details on this feature can be found in Bettini et al. (2020b)). Through the interpreter, which keeps track of the evolved metamodel, our compiler enforces the correctness of the evolution right in the IDE based on the flow of the execution of the migration operations specified by the user. Thus, besides language errors due to types, our compiler immediately detects the wrong usage of Ecore elements in a specific part of the specification, e.g., referring to an element that has been removed or renamed, thus avoiding generating a migrator that would make the migrated metamodels and models invalid.

These mechanisms of Edelta and its DSL allow for fast development cycles since the "live" preview is available even without saving the program in the editor. Moreover, the migrator is generated immediately as part of the compilation, while in COPE, there are a few steps before getting to a migrator (see the COPE tool workflow described above). Thus, with Edelta and its DSL, the time to get to an executable migrator ready to be tested, possibly with automatic testing mechanisms (see our testing strategies in Section 5.1), is very small. Moreover, in COPE, like in our approach, metamodel and model evolution instructions are part of the same migration function but while in COPE one has to manually and properly retrieve all the instances that must be migrated, in Edelta, as shown in Section 3.2, we provide a declarative approach where the developer only has to specify how to migrate instances given some predicates: our migration runtime takes care of the overall migration process. Note that our use of the standard `EcoreUtil.Copier` allows us to seamlessly implement what Herrmannsdoerfer et al. (2009) defines as *coupled transactions*: the migrated model is always valid during the migration, while in Herrmannsdoerfer et al. (2009), during the application of a migration pattern, the model might be in an invalid state (though, in the end, it will be valid). Finally, in Herrmannsdoerfer et al. (2009), only reusable autonomous migration patterns are provided, while, in Edelta, we implement more reusable migration patterns as collaborative or interactive (Section 3.3), that is, also breaking unresolvable changes: the developer only has to specify small code blocks that will be used appropriately during the migration by our reusable migration implementations.

Another approach that is similar to ours is Epsilon Flock (Rose et al., 2010b, 2014b). Edelta and Flock are operator approaches. Flock is a model-to-model transformation language designed explicitly for model migration. It uses a user-controlled conservative copy algorithm, automatically copying the unchanged model elements to the migrated model. The algorithm is user-controlled because the user only specifies the migration strategies related to the evolved elements. For example, if the metaclass `Net` is renamed to `PetriNet`, the user can define the following rule `retype Net to PetriNet`. Flock is based on the Epsilon object language (EOL)[17] and is a rule-based transformation language combining declarative and imperative elements. The language allows for defining migration strategies compactly.

Edelta shares with Flock the ability to automatically copy unchanged model elements and the compactness of migration strategy definitions. Moreover, Edelta can also automatically handle simple metamodel migrations like renaming and deleting. For example, regarding the renaming of `Net` to `PetriNet`, Edelta does not require any model migration strategy from the user.

Flock is effective when the source and evolved metamodels are similar, as mentioned by the authors (Rose et al., 2014b). However, Flock does not allow handling the order in which rules are scheduled and, mainly, does not support simultaneously applying two or more migration rules to the same element since the algorithm uses the first matched rule. These limitations force users to navigate the model to define a more general migration rule. For example, in the ExamXML example described in Fig. A.11, the `OpenElement` is split into two classes `OpenElement_1` and `OpenElement_2`. Therefore, the following rules could be defined: `retype OpenElement to OpenElement_1 when: (original.specificQuestion1 != null)` and `retype OpenElement to OpenElement_2 when: (original.specificQuestion2 != null)`. However, if both the attributes `specificQuestion1` and `specificQuestion2` are

[16] https://groovy-lang.org/

[17] https://www.eclipse.org/epsilon/doc/eol/

set, the algorithm only uses the first rule. Thus, the user cannot define a rule that creates both `OpenElement_1` and `OpenElement_2` model elements. In this case, the user must define a migration rule for the root element model, i.e., `Exam`.

With Edelta, the developer does not have to handle the entire traversal of the model graph to migrate the parts that must be adapted according to the evolved metamodel. It is enough to interleave the metamodel evolution operations with calls to modelMigration passing a predicate and a code block. Since these are code blocks and not expressions, they will be evaluated by Edelta during the co-evolution at the right moments, relieving the developer of handling the internal details of model migrations.

Moreover, our model migration specifications are not based on pattern-matching rules but just predicates on the elements of the metamodel being evolved (for which we provide a few utility functions, as shown in Section 4). This means there is no risk of several possible matching patterns requiring some decision on which to apply and in which order. Edelta applies the migrations in the order specified by the developer, one by one, one after the other. Thus, in Edelta, we do not have the situation described above for Flock when two rules match.

For example, regarding the above-mentioned ExamXML example, in Edelta, the user can decide a migration rule for splitting the OpenElement metaclass to OpenElement_1 or OpenElement_2 exclusively depending on the specificQuestion1 attribute value (see the first listing in Appendix A.2), or she can create OpenElement_1 and OpenElement_2 if specificQuestion1 and specificQuestion2 are both set (see the second listing in Appendix A.2).

Other approaches use different technologies to discover model migration strategies, e.g., search-based approaches (Williams et al., 2012) or feature-based approaches (Di Ruscio et al., 2016). Kessentini et al. (2018) proposes an interactive multi-objective approach that dynamically adapts the model in response to metamodel evolution and interactively suggests edit operations to developers. In this case, the supported model operations must be encoded to enable the algorithm to apply the operators.

9. Conclusions

When a metamodel evolves, all the related artifacts must be adapted in case the applied evolution has broken their validity. This activity of co-evolution, in the specific case of corrupted models, can be also called migration.

Existing approaches propose automatism to deal with this problem with limited cross-domain reuse, due to the domain-specificity of the adopted migration strategies.

In this paper, we first analyzed the literature in order to understand if migration strategies can be shared across different domains. From this analysis, we deduced that, effectively, some migration strategies recur across domains and then could be considered recurring migration patterns.

The Edelta framework, previously presented as a metamodel evolution approach, has been extended to enable the implementation of model migration operations coupled with metamodel evolutions. Using the provided Edelta library, models can be safely migrated simply by reusing the provided functions. The Edelta language also supports the definition of new patterns or the customization of existing ones. We offered a general discussion on the level of automation the framework can reach in co-evolving models and metamodels.

Future work in this direction includes a more extended user study in which we evaluate the real usage of this Edelta application. Metrics can be used to estimate the usability and efficacy of an instrument like Edelta in model migration.

CRediT authorship contribution statement

Lorenzo Bettini: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Writing – original draft, Writing – review & editing. **Amleto Di Salle:** Conceptualization, Data curation, Investigation, Methodology, Validation, Writing – original draft. **Ludovico Iovino:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Alfonso Pierantonio:** Conceptualization, Investigation, Methodology, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The links to software and data are available in the paper.

Acknowledgments

This work was partially supported by the PRIN project "T-LADIES" n. 2020TL3X8X, by the COmmunity-Based Organized Littering (COBOL) national research project (PRIN 2022 PNRR program Contract P20224K9EK), by the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - "Advanced Space Technologies and Research Alliance", of the PNRR MUR project CHANGES (PE0000020), Spoke 5 "Science and Technologies for Sustainable Diagnostics of Cultural Heritage", and by the MUR (Italy) Department of Excellence 2023–2027 for GSSI.

Appendix. Edelta code examples

In this section, we show the implementation of the co-evolution of a few examples in Edelta, relying on some of the reusable refactorings shown in Section 4 and other reusable refactorings of our library, which we did not present in this paper.

A.1. Petrinet

In the following listing, we present an example of migration (both for metamodel and model) for the Petrinet example (Fig. A.10) in Edelta extracted from the studies of Rose et al. (2010a,b).

```
use EdeltaRefactorings as refactorings

def addWeightAttribute(EClass c) {
  c.addNewEAttribute("weight", ecoreref(EInt
      )) [
    makeRequired // lowerBound = 1
  ]
}

modifyEcore modifyNet epackage petrinet {
  ecoreref(Net).name = "Petrinet"
}

modifyEcore introducePTArc epackage petrinet
      {
  refactorings.referenceToClass("PTArc",
      ecoreref(Place.dst)) => [
    addWeightAttribute
  ]
  ecoreref(Place.dst).name = "out"
  ecoreref(Transition.src).name = "in"
```

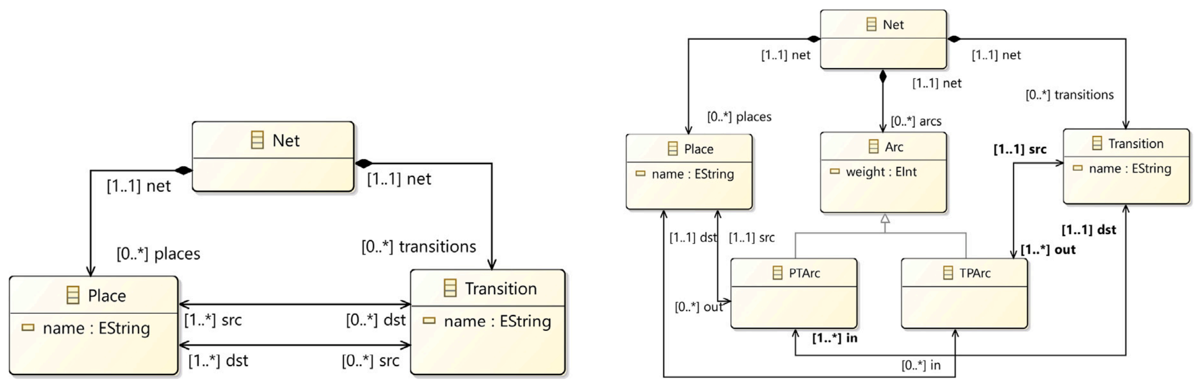


Fig. A.10. Petrinet metamodels - original and evolved.

```
  ecoreref(PTArc.transition).name = "dst"
  ecoreref(PTArc.place).name = "src"
}

modifyEcore introduceTPArc epackage petrinet
     {
  refactorings.referenceToClass("TPArc",
      ecoreref(Transition.dst)) => [
    addWeightAttribute
  ]
  ecoreref(Place.src).name = "in"
  ecoreref(Transition.dst).name = "out"
  ecoreref(TPArc.transition).name = "src"
  ecoreref(TPArc.place).name = "dst"
}

modifyEcore introduceAbstractArc epackage
     petrinet {
  val arc = refactorings.extractSuperclass("
      Arc",
    #[ecoreref(PTArc.weight), ecoreref(TPArc
        .weight)])

  val netRef = arc.addNewEReference("net",
      ecoreref(Petrinet))

  val arcs = ecoreref(Petrinet).
      addNewEReference("arcs", arc) [
    makeContainment
    makeMultiple
    makeBidirectional(netRef)
  ]

  // drop containment from Place and
      Transition
  val placeOut = ecoreref(Place.out)
  placeOut.dropContainment
  val transitionOut = ecoreref(Transition.
      out)
  transitionOut.dropContainment

  // because Arc objects must be contained
      in Petrinet.arcs
  modelMigration[
    copyRule(
      [f | isRelatedTo(f, placeOut) ||
          isRelatedTo(f, transitionOut)],
      [f, oldObj, newObj |
        val migratedNet = getMigrated(oldObj
            .eContainer)
        val migratedArcs = getMigrated(
            oldObj.getValueAsList(f))
        migratedNet.getValueAsList(arcs).
            addAll(migratedArcs)
      ]
    )
  ]
}
```

Besides simple operations, which in Edelta do not require custom model migration strategies, like renaming, in this example, we use “Reference to Class” (Herrmannsdoerfer et al., 2010; Bettini et al., 2022a), which “makes the reference composite and creates the reference class as its new type. Single-valued references are created in the reference class to target the source and target class of the original reference”. Concerning model migration, this is handled automatically in referenceToClass: the original reference is replaced by objects of the reference class, and the features and containment relations are updated accordingly. This model migration does not require any intervention from the user.

We also use “Extract Superclass” with the intended semantics. Also, in this case, the model migration is handled automatically by the Edelta implementation extractSuperclass.

We then introduce the new containment relation for (abstract) arcs and drop the original containment relations from Transition and Place. These metamodel evolution operations require a custom model migration strategy, shown at the end of the listing: we “intercept” the model migration of the original containment references, and, in the migrated model, we add the migrated arcs into the new containment reference arcs. Note that we rely on the getMigrated function to retrieve the migrated versions of the original object (see Section 3.2).

A.2. ExamXML

In the following listing, we present an example of migration (both for metamodel and model) for the ExamXML (García et al., 2012) example in Edelta.

```
use EdeltaRefactorings as refactorings

modifyEcore removeAttributes epackage
     examxml {
  removeElement(ecoreref(question))
  removeElement(ecoreref(optional))
}

modifyEcore introduceExerciseElement
     epackage examxml {
  ecoreref(ExamElement).addNewSubclass("
      ExerciseElement")
}

modifyEcore splitOpenElement epackage
     examxml {
```

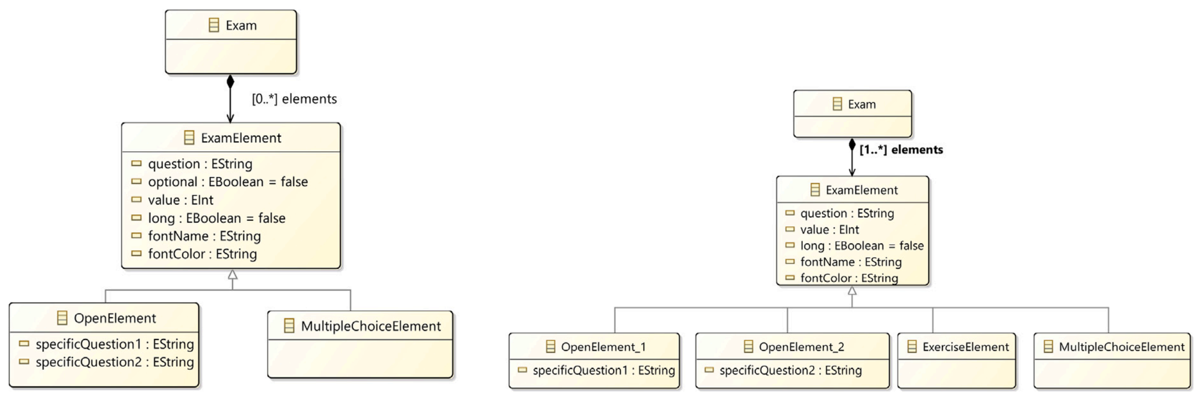


Fig. A.11. ExamXML metamodels - original and evolved.

```
  val toSplit = ecoreref(OpenElement)
  val ePackage = ecoreref(examxml)
  refactorings.splitClass(
    toSplit,
    #["OpenElement1", "OpenElement2"],
    [origElement |
      if (origElement.isSet("
          specificQuestion1"))
        createInstance(ePackage.getEClass("
            OpenElement1"))
      else
        createInstance(ePackage.getEClass("
            OpenElement2"))
    ]
  )
  removeElement(ecoreref(examxml.
      OpenElement1.specificQuestion2))
  removeElement(ecoreref(examxml.
      OpenElement2.specificQuestion1))
}
```

Besides the basic operations of removal of features and the addition of a new subclass, the interesting part is the use of “Split Class” (Herrmannsdoerfer et al., 2010; Bettini et al., 2022a), which splits an existing class (OpenElement in the example) into two classes, all sharing the same features (the original class is removed in the evolved metamodel). In the example, the two obtained classes have to be further evolved by removing a different feature from each of them. The Edelta implementation splitClass also requires a lambda expression used during model migration. In fact, the user has to specify which object to create in the migrated model (i.e., which one of the new classes to instantiate) starting from an object of the original model (i.e., an instance of the original class that has been split and removed). Note that the user only has to create such an instance; splitClass will then take care of migrating the features’ values of the original object (which is a safe operation since all the new classes resulting from the splitting all share the same features). In the above listing, we specify a lambda that creates either an OpenElement1 or an OpenElement2 according to the setting of the features specificQuestion1 or specificQuestion2 in the original object.

Thus, in this implementation, we assume that specificQuestion1 and specificQuestion2 features are mutually exclusively set for each object in the original model. If that is not the case, we could provide a different implementation of the model migration strategy: create an OpenElement1 and an OpenElement2 according to the setting of the features specificQuestion1 and specificQuestion2 in the original object. Thus, if both features were set in the original object, then there should be two objects instead of one in the migrated model. Since this implies that given an object, we create two objects in the migrated model, we cannot easily pass a lambda like the one above to splitClass. However, Edelta also provides an overloaded version of splitClass where you can pass a lambda that takes an EdeltaModelMigrator and can then specify any model migration strategy. That is the version of splitClass used in the following alternative implementation:

```
... // as above
modifyEcore splitOpenElement epackage
    examxml {
  val toSplit = ecoreref(OpenElement)
  val ePackage = ecoreref(examxml)
  val elementsFeature = ecoreref(Exam.
      elements)
  refactorings.splitClass(
    toSplit,
    #["OpenElement1", "OpenElement2"],
    [EdeltaModelMigrator it |
      copyRule(
        wasRelatedTo(elementsFeature),
        [origElementsFeature, origExam,
          newExam |
          val newElements = newArrayList
          val origElements = origExam.
            getValueAsList(
            origElementsFeature)
          for (origElement : origElements) {
            val origElementClass =
              origElement.eClass
            if (origElementClass ==
              getOriginal(toSplit)) {
              if (origElement.isSet("
                specificQuestion1"))
                newElements += createFrom(
                    ePackage.getEClass("
                    OpenElement1"),
                            origElement)
              if (origElement.isSet("
                specificQuestion2"))
                newElements += createFrom(
                    ePackage.getEClass("
                    OpenElement2"),
                            origElement)
            } else {
              newElements += getMigrated(
                origElement)
            }
          }
          newExam.eSet(elementsFeature,
            newElements)
        ]
      )
    ]
  )
  removeElement(ecoreref(examxml.
      OpenElement1.specificQuestion2))
removeElement(ecoreref(examxml.
    OpenElement2.specificQuestion1))
}
```

Here, we use copyRule, which has been presented in Section 4.1. Thus, we "intercept" the migration of the container object and the containment feature (Exam.elements) so that we have full control on the migration of the OpenElement objects. Of course, this alternative implementation of this example requires more work than the first implementation. However, it shows that Edelta provides enough flexibility to the developer for complex refactorings.

References

Anguel, F., Amirat, A., Bounour, N., 2014. Using weaving models in metamodel and model co-evolution approach. In: 2014 6th International Conference on Computer Science and Information Technology. CSIT, IEEE, pp. 142–147.

Barriga, A., Di Ruscio, D., Iovino, L., Nguyen, P.T., Pierantonio, A., 2020. An extensible tool-chain for analyzing datasets of metamodels. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. pp. 1–8.

Beck, K., 2003. Test Driven Development: By Example. Addison-Wesley, ISBN: 0-321-14653-0.

Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A., 2017. Edelta: An approach for defining and applying reusable metamodel refactorings.. In: MODELS. Satellite Events, pp. 71–80.

Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A., 2019. Quality-driven detection and resolution of metamodel smells. IEEE Access (ISSN: 2169-3536) 7, 16364–16376.

Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A., 2020a. Detecting Metamodel Evolutions in Repositories of Model-Driven Projects. J. Object Technol. 19 (2), 14:1–22.

Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A., 2020b. Edelta 2.0: supporting live metamodel evolutions. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. pp. 1–10.

Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A., 2022a. An executable metamodel refactoring catalog. Softw. Syst. Model. 21 (5), 1689–1709.

Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A., 2022b. Supporting safe metamodel evolution with edelta. Int. J. Softw. Tools Technol. Transf. 24 (2), 247–260.

Cadavid, J., Combemale, B., Baudry, B., 2012. Ten Years of Meta-Object Facility: An Analysis of Metamodeling Practices (Ph.D. thesis). INRIA.

Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A., 2008a. Automating co-evolution in model-driven engineering. In: 12th Int. IEEE Enterprise Distributed Object Computing Conf. EDOC 2008, IEEE Computer Society, pp. 222–231.

Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A., 2008b. Meta-model differences for supporting model co-evolution. In: Proceedings of the 2nd Workshop on Model-Driven Software Evolution, Vol. 1, No. 10. MODSE, Citeseer.

Demuth, A., Riedl-Ehrenleitner, M., Lopez-Herrejon, R.E., Egyed, A., 2016. Co-evolution of metamodels and models through consistent change propagation. J. Syst. Softw. 111, 281–297. http://dx.doi.org/10.1016/j.jss.2015.03.003.

Di Ruscio, D., Etzlstorfer, J., Iovino, L., Pierantonio, A., Schwinger, W., 2016. Supporting variability exploration and resolution during model migration. In: Wasowski, A., Lönn, H. (Eds.), Modelling Foundations and Applications - 12th European Conference, ECMFA@STAF 2016, Vienna, Austria, July 6–7, 2016, Proceedings. In: Lecture Notes in Computer Science, vol. 9764, Springer, pp. 231–246. http://dx.doi.org/10.1007/978-3-319-42061-5_15.

Di Ruscio, D., Iovino, L., Pierantonio, A., 2011. What is needed for managing co-evolution in MDE? In: Procs. of the 2nd Int. Workshop on Model Comparison in Practice. IWMCP '11, ACM, ISBN: 978-1-4503-0668-3, pp. 30–38.

Di Ruscio, D., Iovino, L., Pierantonio, A., 2012. Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems. In: Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G. (Eds.), Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24–29, 2012. Proceedings. In: Lecture Notes in Computer Science, vol. 7562, Springer, pp. 20–37. http://dx.doi.org/10.1007/978-3-642-33654-6_2.

Garcés, K., Jouault, F., Cointe, P., Bézivin, J., 2009. Managing model adaptation by precise detection of metamodel changes. In: European Conference on Model Driven Architecture-Foundations and Applications. Springer, pp. 34–49.

Garcés, K., Vara, J.M., Jouault, F., Marcos, E., 2014. Adapting transformations to metamodel changes via external transformation composition. Softw. Syst. Model. 13 (2), 789–806. http://dx.doi.org/10.1007/s10270-012-0297-1.

García, J., Diaz, O., Azanza, M., 2012. Model transformation co-evolution: A semi-automatic approach. In: International Conference on Software Language Engineering. Springer, pp. 144–163.

Hebig, R., Giese, H., Stallmann, F., Seibel, A., 2013. On the complex nature of mde evolution. In: Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16. Springer, pp. 436–453.

Hebig, R., Khelladi, D.E., Bendraou, R., 2017. Approaches to co-evolution of metamodels and models: A survey. IEEE Trans. Softw. Eng. 43 (5), 396–414, ISSN 1939-3520.

Herrmannsdoerfer, M., Benz, S., Juergens, E., 2009. COPE-automating coupled evolution of metamodels and models. In: European Conference on Object-Oriented Programming. Springer, pp. 52–76.

Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G., 2010. An extensive catalog of operators for the coupled evolution of metamodels and models. In: SLE. In: LNCS, vol. 6563, pp. 163–182. http://dx.doi.org/10.1007/978-3-642-19440-5_10.

Iovino, L., Pierantonio, A., Malavolta, I., 2012. On the impact significance of metamodel evolution in MDE. J. Object Technol. 11, 3:1–33.

Kessentini, W., Alizadeh, V., 2020. Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '20, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450370196, pp. 68–78. http://dx.doi.org/10.1145/3365438.3410966.

Kessentini, W., Sahraoui, H., Wimmer, M., 2016. Automated metamodel/model co-evolution using a multi-objective optimization approach. In: European Conference on Modelling Foundations and Applications. Springer, pp. 138–155.

Kessentini, W., Wimmer, M., Sahraoui, H., 2018. Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 101–111.

Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J., Linkman, S., 2009. Systematic literature reviews in software engineering–a systematic literature review. Inf. Softw. Technol. 51 (1), 7–15.

Krause, C., Dyck, J., Giese, H., 2013. Metamodel-specific coupled evolution based on dynamically typed graph transformations. In: Duddy, K., Kappel, G. (Eds.), Theory and Practice of Model Transformations - 6th International Conference, ICMT@STAF 2013, Budapest, Hungary, June 18–19, 2013. Proceedings. In: Lecture Notes in Computer Science, vol. 7909, Springer, pp. 76–91. http://dx.doi.org/10.1007/978-3-642-38883-5_10.

Paige, R.F., Matragkas, N., Rose, L.M., 2016. Evolving models in model-driven engineering: State-of-the-art and future challenges. J. Syst. Softw. (ISSN: 0164-1212) 111, 272–280. http://dx.doi.org/10.1016/j.jss.2015.08.047, URL https://www.sciencedirect.com/science/article/pii/S0164121215001909.

Paternostro, M., Hussey, K., 2006. Advanced features of the eclipse modeling framework. March 2006.

Rose, L.M., Herrmannsdoerfer, M., Mazanek, S., Gorp, P.V., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schätz, B., Wimmer, M., 2014a. Graph and model transformation tools for model migration - Empirical results from the transformation tool contest. Softw. Syst. Model. 13 (1), 323–359. http://dx.doi.org/10.1007/s10270-012-0245-0.

Rose, L.M., Herrmannsdoerfer, M., Williams, J.R., Kolovos, D.S., Garcés, K., Paige, R.F., Polack, F.A.C., 2010a. A comparison of model migration tools. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (Eds.), Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I. In: Lecture Notes in Computer Science, vol. 6394, Springer, pp. 61–75. http://dx.doi.org/10.1007/978-3-642-16145-2_5.

Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A., 2010b. Model migration with epsilon flock. In: International Conference on Theory and Practice of Model Transformations. Springer, pp. 184–198.

Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A., Poulding, S., 2014b. Epsilon flock: a model migration language. Softw. Syst. Model. 13 (2), 735–755.

Rutle, A., Iovino, L., König, H., Diskin, Z., 2020. A query-retyping approach to model transformation co-evolution. Softw. Syst. Model. 19, 1107–1138.

Schmidt, D.C., 2006. Guest editor's introduction: Model-driven engineering. Computer (ISSN: 1558-0814) 39 (2), 25–31.

Steinberg, D., Budinsky, F., Merks, E., Paternostro, M., 2008. EMF: Eclipse Modeling Framework. Pearson Education.

Taentzer, G., Mantz, F., Arendt, T., Lamo, Y., 2013. Customizable model migration schemes for meta-model evolutions with multiplicity changes. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 254–270.

Taentzer, G., Mantz, F., Lamo, Y., 2012. Co-transformation of graphs and type graphs with application to model co-evolution. In: Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G. (Eds.), Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24–29, 2012. Proceedings. In: Lecture Notes in Computer Science, vol. 7562, Springer, pp. 326–340. http://dx.doi.org/10.1007/978-3-642-33654-6_22.

Wagelaar, D., Iovino, L., Di Ruscio, D., Pierantonio, A., 2012. Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In: Hu, Z., de Lara, J. (Eds.), Theory and Practice of Model Transformations - 5th International Conference, ICMT@TOOLS 2012, Prague, Czech Republic, May 28–29, 2012. Proceedings. In: Lecture Notes in Computer Science, vol. 7307, Springer, pp. 192–207. http://dx.doi.org/10.1007/978-3-642-30476-7_13.

Williams, J.R., Paige, R.F., Polack, F.A.C., 2012. Searching for model migration strategies. In: Proceedings of the 6th International Workshop on Models and Evolution. In: ME12, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450317986, pp. 39–44.

Wimmer, M., Kusel, A., Schönböck, J., Retschitzegger, W., Schwinger, W., Kappel, G., 2010. On using inplace transformations for model co-evolution. In: Proc. 2nd Int. Workshop Model Transformation with ATL, Vol. 711. Citeseer, pp. 65–78.

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Shepperd, M.J., Hall, T., Myrtveit, I. (Eds.), 18th International Conference on Evaluation and Assessment in Software Engineering. EASE '14, London, England, United Kingdom, May 13–14, 2014, ACM, pp. 38:1–38:10. http://dx.doi.org/10.1145/2601248.2601268.

Yu, I.C., Berg, H., 2015. A formalisation of analysis-based model migration. In: Hammoudi, S., Pires, L.F., Desfray, P., Filipe, J. (Eds.), MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development. ESEO, Angers, Loire Valley, France, 9–11 February, 2015, SciTePress, pp. 86–98. http://dx.doi.org/10.5220/0005240900860098.

Zhang, H., Babar, M.A., Tell, P., 2011. Identifying relevant studies in software engineering. Inf. Softw. Technol. 53 (6), 625–637.

Lorenzo Bettini is an Associate Professor in Computer Science at DISIA Dipartimento di Statistica, Informatica, Applicazioni 'Giuseppe Parenti', Università di Firenze, Italy, since February 2016. Previously, he was an Assistant Professor (Researcher) in Computer Science at Dipartimento di Informatica, Università di Torino, Italy. His research interests cover design, theory and implementation of programming languages (in particular Object-Oriented languages and Network aware languages). Contact him at lorenzo.bettini@unifi.it, or visit https://www.lorenzobettini.it.

Amleto Di Salle is currently an assistant professor at the GSSI – Gran Sasso Science Institute, L'Aquila - in the Computer Science department. In 2015, he received a Ph.D. in computer science from the University of L'Aquila. His main research activities are related to several aspects of Software Engineering, particularly in distributed systems composition, software architecture, model-based software engineering, and software systems evolution, focusing on technical debt. He has worked on several European and national research projects, such as CHOReOS, CHOReVOLUTION, INCIPICT, Territori Aperti, and Banca Dati Emergenze. He is currently an associate editor of the Journal of Computer Languages and a member of the Journal of Universal Computer Science Editorial Board. He has been involved in the program committee conferences and workshops and organized several workshops, such as MDE4SA@ICSA and FPVM@STAF. Contact him at amleto.disalle@gssi.it, or visit https://amletodisalle.github.io/.

Ludovico Iovino is Assistant Professor at the GSSI – Gran Sasso Science Institute, L'Aquila - in the Computer Science department. His interests include Model Driven Engineering (MDE), Model Transformations, Metamodel Evolution, code generation and software quality evaluation. He has been included in program committees of numerous conferences and in the organization of various conferences, e.g., STAF, MODELS, iCities, he is also part of the steering committee of models and evolution workshop co-located with MODELS. He is part of different academic projects related to Model Repositories, model migration tools and Eclipse Plugins. He is in the editorial board of the COLA journal and in the Review board of TSE. Contact him at ludovico.iovino@gssi.it, or visit http://www.ludovicoiovino.com.

Alfonso Pierantonio is Professor at the University of L'Aquila, Italy. His interests include Model-Driven Engineering with a specific emphasis on co-evolution problems, bidirectionality, and megamodeling. He has chaired a number of international conferences and organized numerous scientific events (including ICMT, STAF and MODELS). He is in the editorial board of several scientific journals (including SoSyM and JOT). Contact him at alfonso.pierantonio@univaq.it, or visit http://pieranton.io.