Software and Systems Modeling
https://doi.org/10.1007/s10270-025-01357-3

TOOL PAPER

Edelta: a versatile framework for migrating clients' EMF models

Lorenzo Bettini[1] · **Amleto Di Salle**[2] · **Ludovico Iovino**[2]

Received: 16 July 2025 / Revised: 13 November 2025 / Accepted: 17 December 2025

Abstract
Metamodels and models are artifacts that must evolve together to maintain consistency and validity within model-driven engineering ecosystems. However, developers often lack control over models created and managed by external clients, making it challenging to ensure model conformance when metamodels evolve in new versions of model-based applications. In this paper, we present an extended version of Edelta, a Java framework and DSL for the co-evolution and refactoring of EMF metamodels and models, which now enables metamodel developers to deliver applications together with migration code that updates clients' models. Our approach supports complex scenarios involving mutually dependent metamodels and models and provides four strategies for integrating migration logic both within Eclipse-based applications and in standalone REST services. We demonstrate the effectiveness of our solution through two comprehensive examples where we apply the four migration strategies introduced. We describe automated testing methodologies for migrating code and discuss the approach in terms of usability and maintainability. With the proposed approach, clients' models are transparently and reliably migrated to conform to new metamodel versions, regardless of their version age or the complexity of their dependencies.

Keywords Metamodels · Co-evolution · Model migration · EMF · Refactoring

1 Introduction

Modeling ecosystems are interrelated modeling artifacts working together to support the development of domain-specific software systems [1]. The defined artifacts are built on top of central models, i.e., metamodels, defining the abstract syntax of the modeling language. When a metamodel evolves, it may trigger a ripple effect on the interrelated artifacts [2, 3], which may be corrupted by applying changes to the metamodel. Changes may have different effects [4] on the artifacts; for instance, a refactoring applied to the metamodel can be transparent to already defined models and may heavily affect a transformation or a code generator [5]. Restoring the consistency of an interrelated artifact with its metamodel is called *co-evolution* and consists of an activity that not only is error-prone if done manually, but it can be expensive in terms of automation to develop [6–9]. The co-evolution of metamodels and models is one of the most investigated activities in literature [10] and still presents many directions to be improved in terms of automation, safety, and performance.

Modelers typically create and maintain metamodels, also building and deploying tools, constituting the ecosystems. For this reason, we also define them as "developers" in this context, and we intend by "clients" the "modelers" who use such tools to create and maintain models.

Indeed, when a metamodel evolves, developers of the metamodels have little control over all the models created, edited, and manipulated by external clients. In this paper, we deal with the problem of keeping clients' models consistent with evolved versions of model-based applications. The Eclipse Modeling Framework (EMF) is one of the most popular workbenches for defining MDE applications [11, 12], and in this paper, we base our entire approach on this technical space. We tackle the typical scenario in which the developers provide an "application" consisting of Eclipse plug-ins or an RCP product built on top of EMF for the model management and the UI (e.g., editors based on trees, tables, etc., for editing model files); the software is deployed and distributed

Communicated by Abel Gómez and Jordi Cabot.

✉ Lorenzo Bettini
lorenzo.bettini@unifi.it

✉ Ludovico Iovino
ludovico.iovino@gssi.it

Amleto Di Salle
amleto.disalle@gssi.it

[1] DiSIA, University of Florence, Florence, Italy

[2] Gran Sasso Science Institute, L'Aquila, Italy

Published online: 24 February 2026

on an update site, and clients install the plug-ins in their Eclipse instance. When a metamodel evolves, a new version of the application must be released, and existing client models may become corrupted. Even small modifications may lead to potential data loss. Existing frameworks often implement recovery strategies based on heuristics that may maximize deletion in exchange for automation, or minimize information loss in exchange for human intervention [13]. Being in control of the migration strategy, dealing with clients' models, with proper tools and automations, is therefore essential to ensure a steady co-evolution of model-based ecosystems and to minimize human intervention.

Edelta is a framework and a DSL designed for the refactoring and evolution of EMF metamodels and models [14–16]. In a nutshell, Edelta differs from other metamodel evolution approaches by treating metamodel evolution as safe, reusable refactorings rather than ad hoc transformations. It provides a dedicated DSL with built-in consistency checks, live IDE feedback, and dependency-aware evolution, enabling developers to evolve metamodels interactively and safely, unlike previous tools that focus mainly on offline transformations or model migration. Edelta's original approach to co-evolution required all relevant artifacts (metamodels and models) to be available during the co-evolution process. This assumption may be in contrast with real-world scenarios, where client models are distributed and not under the control of the developers at the time of evolution. To address this limitation, in [17] we extended Edelta and its DSL with mechanisms to support the migration of client models, even when these models are unavailable during metamodel evolution. With this extension, developers can write concise migration code that is executed on the clients' machines, ensuring that models are automatically updated to conform to the new metamodel version.

This paper significantly extends the work presented in [17] in several key aspects:

- We further extend Edelta and its DSL to support the migration of client models in the presence of mutually dependent metamodels and models, addressing more complex co-evolution scenarios, involving dependent metamodels and models, complementing the example from [17].
- We present additional strategies for integrating model migration into clients' Eclipse applications, providing practical guidance for real-world adoption, including an application outside the Eclipse ecosystem, in a standalone REST service.
- We detail a methodological workflow for using Edelta, from specifying co-evolution scenarios and implementing migration code, to applying migrations in client environments (both Eclipse-based and REST-based). In particular, we describe automated testing methodologies for validating migration code (unit tests) and its integration within client applications (functional UI/integration tests), leveraging the new testing libraries provided by Edelta.
- We provide an in-depth discussion of the lessons learned from applying our approach, highlighting both its strengths and potential limitations.

We demonstrate our approach and tools using EMF to generate the Java code for the model and the editor. We will implement a few metamodel–model refactorings/evolutions in Edelta and the minimal Java migration code for migrating models in clients' applications. The clients of our application are meant to install our deployed features and plug-ins in their existing Eclipse installation, create projects in their workspaces, and use our EMF editors to create and modify model files compliant with our Ecore models. The clients will then be able to install the new version of our application, which will contain the new metamodels and the migration code. Depending on the implemented strategy, the check and possible needed migration of the models will be performed automatically by the application, or the clients will be able to trigger the migration manually. The same migration code can also be used in a REST service, implemented in Java, with Spring Boot.

In the examples shown in this paper, we will implement several metamodel/model refactorings/evolutions in several separate steps, each one corresponding to a new version of the application. These separate steps are meant to simulate the development lifecycle of an EMF application, where the metamodel evolves in time together with the application. One can think of each step (i.e., each new released version) taking place after a few months/years. Having several refactorings/evolutions in the examples will allow us to show the small maintenance effort to introduce a new migration after creating the whole migration infrastructure.

We assume the reader is familiar with the basic usage of EMF, its API, and generation mechanisms (i.e., to create Java model code and UI code). Moreover, some basic familiarity with Eclipse plug-in development and REST services is also required.

The paper is structured as follows: Sect. 2 briefly summarizes Edelta and describes the problem that is the subject of this paper; it also introduces the approach to address such a problem. Section 3 fully describes how to handle the migration of clients' models with Edelta with two running examples; it also shows a few possible strategies to use the migration logic in an Eclipse application and in a REST service. Section 4 discusses the features and possible drawbacks of the strategies for model migrations and shows some performance measures to assess the usability of the approach. Sect. 5 presents related works, and Sect. 6 concludes the paper.

2 Dealing with Metamodel/models Co-evolution with Edelta

Edelta is an open-source framework designed for the refactoring and evolution of EMF metamodels and models. Edelta consists of a Java runtime library (a Java API built on top of the standard EMF API, with a fluent style and stronger static typing than the EMF one) and a DSL to specify basic metamodel changes (e.g., additions and deletions) and complex reusable changes by aggregating existing library refactorings. It is available at the following GitHub repository https://github.com/LorenzoBettini/edelta.[1]

Edelta DSL and Eclipse IDE integration The Edelta DSL (implemented with Xtext [18], the most popular Eclipse framework for the development of programming languages and DSLs), provides an integration with the Eclipse-based IDE with a rich set of features, such as syntax highlighting, content assist, code navigation, quick fixes, incremental building, error reporting, and debugging. In particular, the Edelta Eclipse editor offers a live view on the evolution of the metamodels, while the user is writing an Edelta program. Moreover, the Edelta DSL is statically typed, relying on type inference so that most types can be omitted in declarations, and also statically checks the correct usage of the metamodel elements used in the Edelta programs. It is also completely compliant and interoperable with Java's type system, so an Edelta program can seamlessly use any existing Java code and Java libraries. Edelta was first presented in [19], and an updated version was subsequently presented in [14].

We briefly summarize the main features of the Edelta DSL, which are relevant to this paper. An example of an Edelta program is shown in Listing 1.

Listing 1 Example of an Edelta program

```
1 migrate "http://www.mypackage.org/v1" to "http://www.
      mypackage.org/v2"
2 migrate "http://www.othermypackage.org/v1" to "http://www.
      othermypackage.org/v2"
3 ...
4 use ... as ...
5 ...
6 def <name>(<... parameters ...>) : <returntype> { <body> }
7 ...
8 modifyEcore <name> epackage <EPackage name> { <body> }
9 ...
```

In the previous versions of the Edelta DSL, metamodels were imported using their package names. This would not allow us to identify metamodel versions. In [17], we introduced the `nsURI` mechanism in the Edelta DSL to refer to metamodels and their versions. References to metamodels subject to evolution are specified by their `nsURI`, the standard EMF mechanism to univocally refer to an `EPackage` by an identifier containing information about a version.[2] Thus, in the Edelta DSL, we refer to metamodels and their evolution with this syntax: `migrate "<nsURI of the old version>" to "<nsURI of the new version>"`.

The DSL compiler checks that a metamodel (i.e., an `.ecore` file) with the specified `nsURI` exists in the classpath and checks that the developer also specifies a new `nsURI`, which will identify the new metamodel after the Edelta evolution operations specified in the program are applied. For simplicity and clarity, in this paper, we will use `nsURI`s ending with `v1`, `v2`, etc., to specify versions of metamodels, although the `nsURI` can be freely specified as long as it is unique within the same application.

The rest of the Edelta program continues with the optional import of existing Edelta libraries with the syntax `use... as...` to be used in the current program.

Reusable functions can be defined with `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. Actual evolution operations on a specific imported `EPackage` are 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, and it should be easily understood by Java developers. The Edelta compiler translates Edelta programs into standard Java code, which relies on the Edelta runtime library. For all the features of Edelta, its DSL, and its Eclipse-based IDE, we refer the interested reader to the Edelta website and to [14, 16].

It is important to stress that while the Edelta DSL is meant to be used from Eclipse, thanks to its integrated IDE support, the generated Java code and the Edelta runtime library can be used in any Java application, including headless applications, such as REST services (as we will see in Sect. 3.4). Thus, the generated Java code only depends on the Edelta runtime library and the standard EMF API, which are both independent of Xtext and Eclipse.

Library of metamodel refactorings

A catalog of metamodel refactorings implemented in Edelta was presented in [15]. Then, the catalog has been extended to also handle the co-evolution of metamodels and models in [16].

Our refactorings library functions take care of both the metamodel evolution and the corresponding model migra-

[1] The source code can also be found on Zenodo at https://doi.org/10.5281/zenodo.17582569.

[2] For example, EMF uses `http://www.eclipse.org/emf/2002/Ecore` for the `nsURI` of `Ecore`, thus using the year for the version information (since then, Ecore has not changed).

tion. Depending on the specific refactoring, the library function will require some parameters or code to allow the function to perform the model migration correctly. In that respect, the library organizes migration patterns into three categories based on their automation capabilities and requirements. *Autonomous patterns* can fully automate both the metamodel evolution and model migration without any external intervention, as they preserve all information and have clear transformation rules. *Collaborative patterns* require some degree of developer intervention in the model migration process implementation, typically to provide missing information or resolve ambiguities that cannot be automatically inferred. *Interactive patterns* involve more substantial user participation (e.g., by prompting the user when a specific part of the model is being migrated; the user is asked how to set some specific values for the migrated object based on the values of the original object), often requiring design decisions or domain-specific knowledge that cannot be automated. This classification helps developers understand the level of automation they can expect when applying each pattern and guides them in planning the migration effort required for their specific metamodel changes.

Note that some refactorings have opposite operations, which can be used to migrate models back to a previous version of the metamodel if needed. For example, `subclassesToEnum` and `enumToSubclasses` (used in Sect. 3.2.2) are reverse operations of each other; the same holds for `classToReference` and `referenceToClass` (used in Sect. 3.3.1). This reversibility feature is important because it allows users to easily undo certain structural changes if they decide the refactoring was not beneficial, which is particularly useful during exploratory refactoring sessions.

In any case, the developer can also specify custom model migration logic for specific scenarios. For example, when an optional feature becomes mandatory, the developer will specify how to set the value of such a feature to a default value if it was not set in the original model. This control over the migration process is also crucial for complex refactorings where the model migration logic cannot be fully automated, since it may depend on domain-specific knowledge. An example of such a scenario is shown in Sect. A.

Runtime API for model migration

In [17], we extended the Edelta runtime library to support the migration of clients' models from an older version of a metamodel to a newer one. In that respect, the entry point for the developer is the class `EdeltaVersionMigrator`, which the developer configures with a few information: the current (i.e., latest) `EPackage` instances and the Edelta programs (i.e., references to Java classes generated by the Edelta DSL compiler) responsible for migrating metamodels and models from a specific version to the next one. With such a configuration, the `EdeltaVersionMigrator` will automatically scan the models in some given paths to check whether they need to migrate from one version to the next. The check and possible migration will be reiterated until all models are up-to-date, i.e., conformant to the current versions of the metamodels. In the rest of the paper, we will demonstrate how to use the Edelta DSL to write refactoring/evolution programs and how to use the `EdeltaVersionMigrator` class to migrate models from older versions to the latest one.

The migration mechanism

The Edelta programs contain references to the Ecore files representing the metamodels to evolve. All the Ecore files representing all the versions of the metamodels must be part of the application's code (technically, they must be found in the classpath). Such files must not be accessible to clients; they are needed only by the migration code. This is expected: since we must load clients' models that are "stale" concerning the current version of our application's metamodels, EMF needs the exact version of the `EPackage` referred by the clients' models. The main constraint of this approach is that the application's deployment bundle grows with each metamodel version. However, since Ecore files are typically small (a few KB each), this overhead is negligible in practice, even after multiple evolution cycles. As textual files, Ecore files will also be efficiently compressed in the final deployment JAR file of the application. Moreover, this approach enables migration from any historical version to the current one.

By design [16], Edelta migrates metamodels (Ecore files) and models (XMI files) together. Since, in this context, the developer owns the Ecore files, and the clients own the model files, we need to use the exact Ecore version of the clients' models to migrate them from an older version to the next one. This co-evolution, taking place on the client's computer (workspace), will only save XMI migrated files and not effectively generate new versions of the Ecore files: these are needed only temporarily, in memory, to perform a correct co-evolution. Such requirements are automatically implied by using the Edelta DSL and the project generated by our wizard.

Moreover, during the iteration of the migration process described above, the `EdeltaVersionMigrator` will only operate on in-memory EMF resources. Model files will be saved to the disk only after all the necessary migration steps have been applied. No intermediate saving of models is performed. Otherwise, models could be left in an inconsistent state if a failure occurs during the migration process. In case of any failure, the entire migration process is aborted, and no model file is modified on the disk. The migration process logs several information about the migration steps performed and any issues encountered, which can be useful for debugging purposes. As usual, in migration operations, it is recommended to back up the models before performing any migration. In particular, storing models under a version

control system (e.g., Git) is a good practice to revert to previous versions if needed. In our library, we do not provide any rollback mechanism since it is orthogonal to the migration process itself and can be easily implemented by the developer if needed.

The developer will deploy the "migration" code as part of the application. Our main goal is that the application's clients will not have to perform any specific, difficult manual operations besides updating their Eclipse installation with the new versions of the application, as it is standard in the Eclipse ecosystem.

2.1 Motivating example

In this section, we illustrate, through an example, how applying evolution to a metamodel can impact the existing artifacts, particularly the models developed by clients.

The example's source code, in which the metamodels have been previously used in [19, 20], can be found online at https://github.com/LorenzoBettini/edelta-example-migration-personlist[3] and the workflow used in this motivating example is summarized in Figure. 1.

We implement a standard EMF project (step ❶), based on a single Ecore file (`Persons.ecore`).[4] We then generate the corresponding Java model code, the `.edit` and the `.editor` plug-in projects using the EMF generator based on the GenModel file (Fig. 2) in step ❷. The `nsURI` of the metamodel is `http://edelta/PersonList/v1`.

As it is standard in Eclipse plug-in development, we start an Eclipse application from the "development workspace" to test our plug-ins manually. This will start another Eclipse instance where the plug-ins of the development workspace are installed (step ❸). We will call this other Eclipse instance the "client instance" and the corresponding workspace the "client workspace" since they simulate the client's environment. Moreover, throughout the examples, each time we modify our Eclipse plug-ins in the development workspace, we restart the "client instance". It can be seen as the simulation of i) the deployment of an updated version of our EMF application and ii) the update of the client's Eclipse.

In the client workspace (Fig. 3), we create an Eclipse project, use the EMF generated wizard to create a `.persons` model file (step ❹), and edit it with the corresponding "Persons tree editor". For demonstration purposes, we have two model files in this workspace.

Going back to the development workspace, we use the Edelta project wizard to create an Edelta project (an Eclipse plug-in project with some source folders and dependencies already configured) that is used to evolve the metamodel subject of this example (step ❺). We call this project the "migrator" project.

We copy the `Persons.ecore` from the main EMF project into the `ecoreversions` source folder of the migrator project. In particular, we copy the file into a subfolder with a sensible name, e.g., `v1`, reflecting that the contained file is the first version of the Ecore. Note that the names of the Ecore files and the directories are not important.[5]

The `ecoreversions` source folder is meant to store all the "past" versions of the metamodel of our EMF application. The `.ecore` files in this source folder will be copied into the final JAR of the project. In fact, the Edelta runtime for the migrator will load Ecore files for previous versions from the classpath, through a `ClassLoader`. This is crucial to execute migrations not only in an Eclipse environment, but also in any Java application, like the REST service presented in Sect. 3.4.

In the migrator project's main source folder, we create an Edelta file (extension `.edelta`) to dictate how the metamodel must evolve (step ❻). Since this Edelta program is meant to migrate the Ecore from version 1 to version 2, we call this file `FromV1ToV2.edelta`. We write the Edelta program in Listing 2 to perform a first simple refactoring: we rename the two attributes of the person using a capital letter. While the renaming is a simple but quite diffused and cited refactoring [16, 22] in metamodel evolution examples, it can be interleaved with many other evolutions, aggravating the entire co-evolution strategies to be undertaken. We use `ecoreref` to refer to the metamodel's elements and the syntactic sugar for setter methods (i.e., `.name = "..."` is equivalent to `.setName("...")`).

Listing 2 The Edelta program for the first refactoring.

```
// file: FromV1ToV2.edelta
migrate "http://edelta/PersonList/v1" to "http://edelta/
    PersonList/v2"

modifyEcore renameAttributes epackage persons {
  ecoreref(firstname).name = "firstName"
  ecoreref(lastname).name = "lastName"
}
```

[3] The source code of the example can also be found on Zenodo at https://doi.org/10.5281/zenodo.17294752.

[4] Edelta and its DSL also support the co-evolution of several interdependent metamodels, as shown in [21], and so does this extended version, as described in Sect. 3.3.

[5] In the previous version [17], subdirectories were not supported: we had to store each version of Ecore files with unique names. In this extended version, we extended Edelta to handle subdirectories because the names of the Ecore files are crucial to stay the same as the original ones when we have Ecore files with dependencies, as in Sect. 3.3. In fact, an `.ecore` file refers to another `.ecore` file in the same project through the file name.

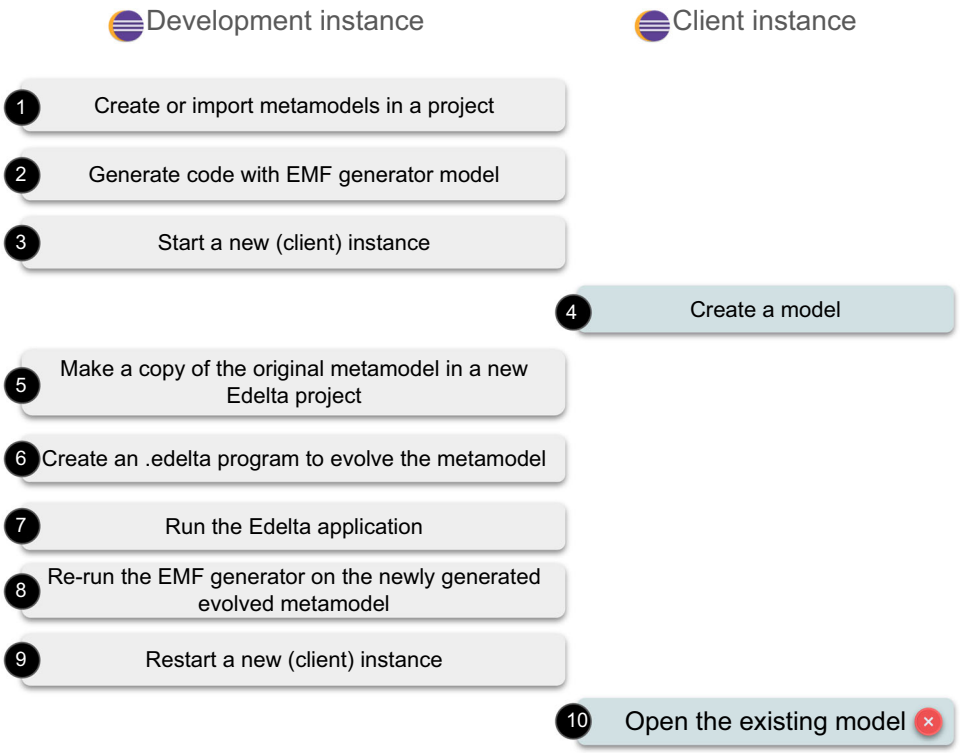


Fig. 1 Summarized workflow described in the motivating example

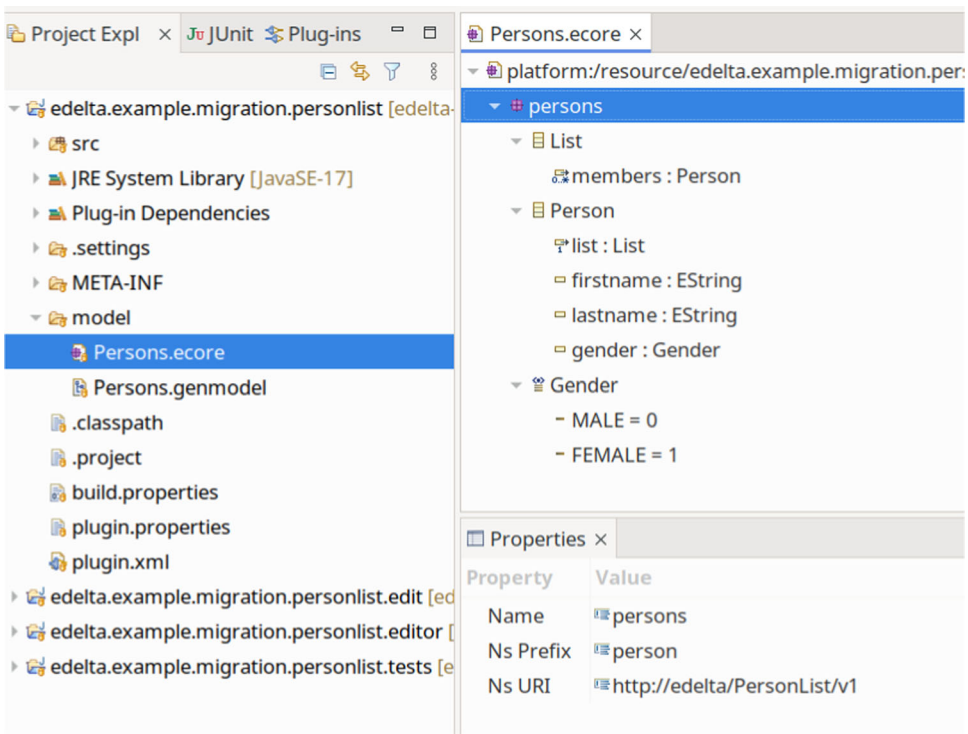


Fig. 2 The initial state of the EMF application

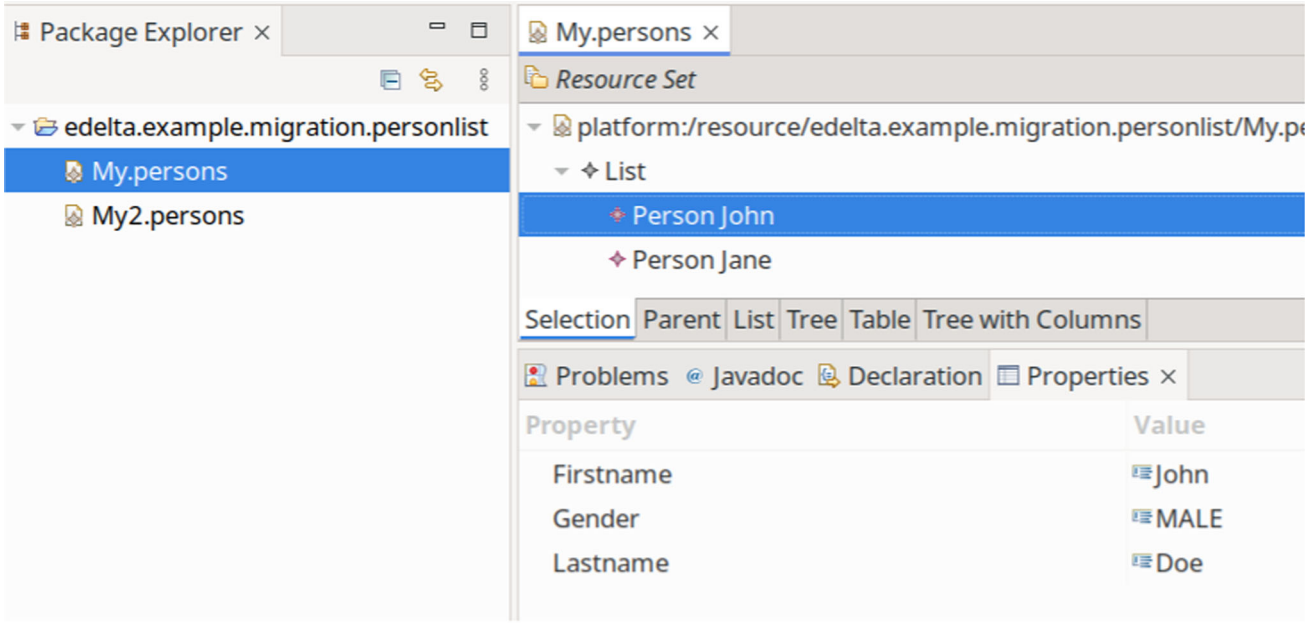


Fig. 3 The editor of our EMF application in the "client workspace."

The Edelta compiler links the namespace URI specified after `migrate` to the Ecore file `v1/Persons.ecore`, by relying on the Xtext index capabilities for Ecore files. Of course, if no Ecore file can be found in the project with the corresponding namespace URI, the compiler will issue an error. Moreover, the Edelta editor provides code completion for the namespace URI, according to the Ecore files in the project, so there is no need to write it manually.

Note the outline in Fig. 4 showing the resulting Ecore (in particular, the outline shows modified/new elements of the Ecore under evolution in boldface; it also selects the node in the outline corresponding to the line in the editor corresponding to the change).

The Edelta compiler generates the corresponding Java class into `edelta-gen`. When applying simple operations to a metamodel, like removals and renamings, the Edelta runtime library automatically handles the corresponding operations for migrating the models.

We run the Edelta program (step 7), which loads the referred Ecore file `v1/Persons.ecore`, applies the specified changes, and saves the resulting Ecore into the folder `modified`. The `modified` directory is meant to be ephemeral and only stores temporarily generated files.

After verifying that the evolved/refactored metamodel is as intended, we copy this Ecore file into the main EMF project and re-execute the EMF generator so that the model Java code and the editor code are updated according to this new version of the metamodel (step 8).

On a side note, Edelta does not handle the GenModel and the standard EMF generation mechanisms, which are orthogonal to Edelta. In particular, all the references to the Ecore elements in the Edelta programs do not need any Java-generated model code: we refer directly to Ecore elements. Thus, possible customizations to the EMF-generated code, calling directly Java methods or classes related to our renamed refactorings, should be handled and updated manually (based on the Java compilation errors). No further intervention will be required if we do not heavily customize the EMF-generated Java code in an EMF application.

Thus, our EMF application has been updated with a refactored metamodel and the updated EMF-generated Java code. As such, if clients update to this new version, their models will no longer be readable and editable by the new version of the editor. We can verify that by restarting the "client instance" in (step 9) and opening one of the model files in (step 10). As expected, we get an error: the model refers to the old `http://edelta/PersonList/v1` and the references to name attributes are not valid anymore (Fig. 5).

2.2 Migrating clients' models with Edelta

In this subsection, by using the running example, we detail our development approach for migrating clients' models after an evolution of the metamodel with Edelta. This approach is designed to ensure that model migration is robust, repeatable, and as automated as possible, and it consists of several key steps, each contributing to a reliable migration workflow:

⇒ *Implement a migration program in Edelta:* First, we write an Edelta program that specifies how the metamodel

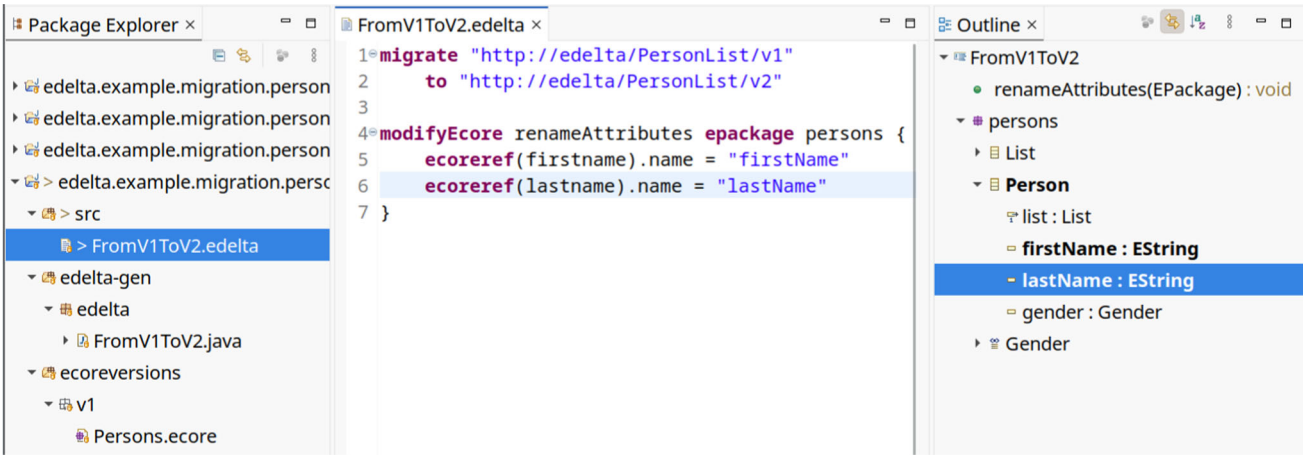


Fig. 4 The first Ecore evolution with Edelta

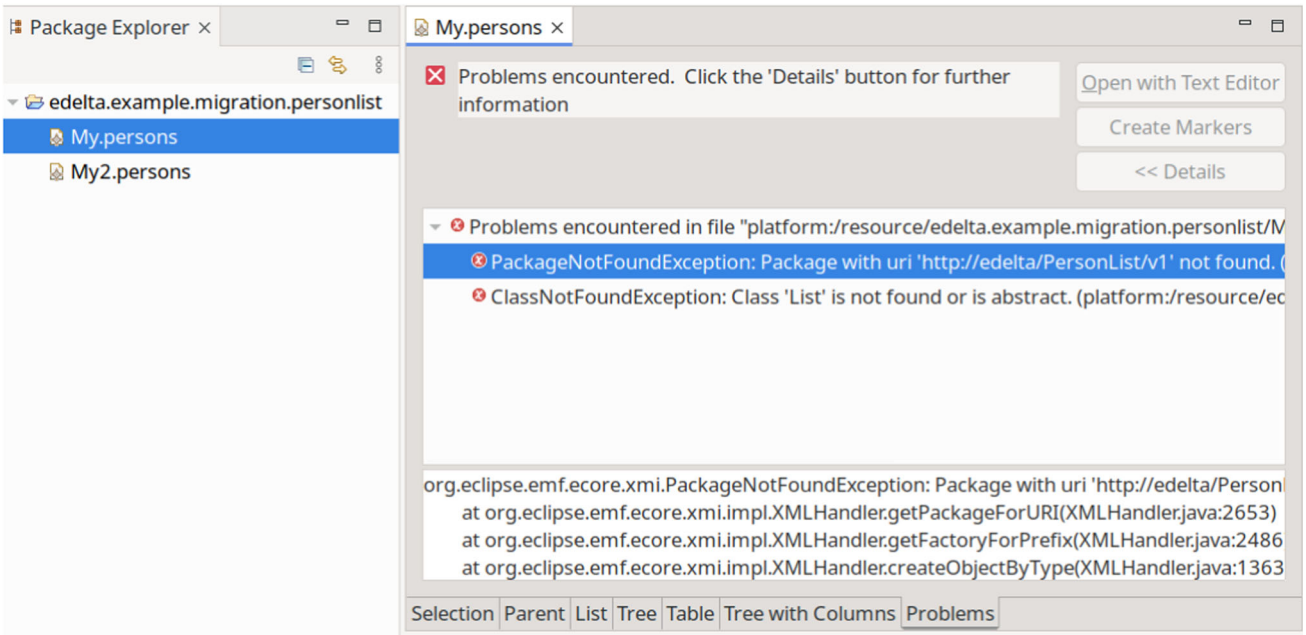


Fig. 5 The client's models are not valid anymore

should evolve and how the corresponding models should be migrated, as shown above. This program leverages the Edelta DSL to describe both simple and complex changes, ensuring that all necessary transformations are captured in a type-safe and declarative manner. By using the Edelta DSL, we can take advantage of its integration with Java, its static typing, and its support for reusable migration logic.

⇒ *Implement a Java class to run the migration:* Next, we implement a Java class that utilizes the Edelta runtime library to execute the migration program on the client's models (Sect. 2.2.1). This class is responsible for configuring the migration process, such as specifying the current (i.e., latest) version of the metamodel, registering the migration steps (i.e., the Java code generated by the Edelta DSL compiler), and identifying the model files to be migrated. It is important to note that this class is written after the first migration program is available, as it depends on the Java code generated by the Edelta compiler from the Edelta DSL. For subsequent metamodel evolutions, only minimal updates to this class are required, typically involving the registration of new migration steps.

⇒ *Write unit tests for migration logic:* To ensure the correctness of the migration, we develop unit tests that exercise the migration program (Sect. 2.2.2). These tests use representative input models conforming to previous versions of the metamodel and verify that, after migration, the models conform to the latest version and meet

all expected properties. Unit tests are essential for catching migration issues early and for supporting regression testing as the migration logic evolves. For subsequent metamodel evolutions, only minimal updates and additions to unit tests are required.

⇒ *Integrate and validate migration in the EMF application:* Finally, we integrate the migration mechanism into the EMF application so that it can be invoked by clients in a transparent and user-friendly way. This part is shown in Sect. 3. This integration can take different forms, such as triggering migration when a model is opened in the editor, providing an explicit migration command in the UI, or running migration automatically on workspace startup. To validate this integration, we recommend implementing UI functional tests (for example, using SWTBot) that simulate user interactions and verify that the migration is correctly performed on client models in realistic scenarios (Sect. 3.2.1). These tests help ensure that the migration process is not only technically correct but also seamlessly integrated into the user's workflow. For subsequent metamodel evolutions, these UI tests should not require any maintenance.

By following our approach, developers can confidently evolve their metamodels and applications, knowing that client models will be reliably migrated to remain consistent with the latest version.

We emphasize that our approach assumes the initial models are valid according to the metamodel version they conform to and that they do not contain specific user-defined extensions breaking the metamodel constraints. Validation errors on models are reported by EMF in the Eclipse workspace and in the EMF editors. Thus, clients are aware of such errors when editing their models. Such errors must be fixed by the clients before performing the migration. If the migration is performed on invalid models, the migration process may fail and be aborted without modifying any model file, or it may produce invalid models according to the latest version of the metamodel. In general, for initial invalid models, it is not possible to guarantee that the migration will produce valid models.

As hinted above, this approach supports extensibility and maintainability, as new migrations can be added incrementally with minimal changes to the existing infrastructure. In fact, when adding a new migration, the existing infrastructure only requires the following minimal updates:

- a new instruction in the Java class responsible for running the migration to register the new migration;
- new unit tests to cover the new migration;
- update the expectations of existing unit tests according to the new latest version of the metamodel. Since all unit tests have the same expectations concerning the latest version of the metamodel, such expectations must be updated in a single place only.

Listing 3 The class responsible for migrating the client's models.

```
public class PersonsModelMigrator {
  public Collection<Resource> execute(String modelPath) throws Exception {
    EdeltaVersionMigrator migrator = new EdeltaVersionMigrator();
    // load the current (latest) version of EPackage
    migrator.loadCurrentEPackage(PersonsPackage.eINSTANCE);
    // register the Edelta migration code
    migrator.registerMigration(FromV1ToV2::new);
    // load the models to check and migrate
    migrator.addModelFileExtension(".persons");
    migrator.loadModelsFrom(modelPath);
    // let the Edelta framework do the rest
    return migrator.execute();
  }
}
```

In particular, the existing UI tests simulating user interactions with the EMF application do not require any maintenance, unless we decide to change the UI itself, which is orthogonal to the migration process.

2.2.1 Implement a model migration

Using the new Edelta `EdeltaVersionMigrator`, we implement in the migrator project the class responsible for migrating the client's models (Listing 3). This class is meant to implement the entry point for calling the Edelta program(s) that migrate the models from the previous version to the current one. Such a code cannot be generated by our Edelta compiler because it refers to information that is not in the Ecore files, such as the path of the models to migrate, the file extensions for the model files, and the current version of the `EPackage`. As we will see in the following, most of the code must be written only once, and it can be reused for further evolutions of the metamodel with a minimal update.

We need to perform the following steps after creating the `EdeltaVersionMigrator` instance:

1. We load the current version of the `EPackage`: we do that by using the Java code generated by EMF, `PersonsPackage.eINSTANCE`; This is meant to always refer to the current (i.e., most up-to-date) version of the EMF Java code, so, unless we heavily refactor our EMF application, by also changing the "name" and "prefix" of the metamodel, it will stay valid even after further evolutions and re-generations.
2. We register the Java code generated by the Edelta DSL compiler; `FromV1ToV2` is the Java class generated from the Edelta program shown above (we pass a "constructor reference", since the actual instantiation is handled by the Edelta runtime at a later stage).

3. We specify the file extension of our EMF application models (by default, Edelta handles automatically ".xmi" files only). The migrator will then look for all the files with this extension in the specified path and load them as EMF resources. It will also automatically ignore and skip other files with different extensions.
4. We load the models in the specified path.
5. We run the `EdeltaVersionMigrator` instance, which will scan the models and detect which ones must be migrated according to the information the Edelta runtime gathers from the specified information. The instance will also return the collection of the effectively migrated EMF resources. This information can be useful in the UI to show some reports to the user.[6]

In this example, we decided to migrate all the models in the specified path. In particular, as we show in the following (Sect. 3.1), it will be the path of the current project of a client's workspace. Of course, the same code can be used to migrate all the model files in all the projects of the client's workspace, one project at a time (Sect. 3.2). The path can refer to any physical location, not necessarily related to the Eclipse workspace (as we will show in the REST service of Sect. 3.4). This strategy is left to the developer, who can also decide to migrate only a file, or to migrate projects in several paths (i.e., using `loadModelsFromPaths` that accepts several paths). Migrating all the models together has the benefit that possible interdependent models will be migrated together, which is a requirement for their validity.

2.2.2 Unit tests

Before using this class in our EMF application, it is best to test it with a JUnit test. We should have a model for the first version of the Ecore, use it as an input, try to migrate it with the `PersonsModelMigrator` we have just written, and compare the result to verify it is as expected. See Fig. 6.

For such unit tests, we follow this approach:

1. We use a valid XMI file for the first version of the Ecore and store it in a subdirectory of the `inputs` directory (in a subdirectory reminding the version of the model);
2. We copy the same file in the `expectations` directory;
3. We run the test expecting it to fail (the migrated model must be different);
4. We check the differences to manually verify the correct migration;
5. We copy the migrated models from `output` to `expectations` and verify the test now succeeds.

For subsequent migrations (see the next sections), we will have to update the `expectations`, which is meant to contain the latest version of the migrated model(s), according to the most recent version of the evolved Ecore. Maintaining such tests requires a minimal amount of manual tasks. We will add new test cases, without removing the existing ones: all the test cases will "prove" we can migrate from any previous versions to the latest one. Since these JUnit tests do not require a running Eclipse, they are as fast as unit tests are expected to be.

Note that we also use the utility method `assertResourcesAreValid`.[7] This will run the EMF validator (using the `Diagnostician.INSTANCE`) and check that there are no validation errors. During loading and saving models, EMF already automatically checks that models are valid according to base constraints (e.g., assigned values are correct with respect to types and that there are no dangling references). However, it does not perform additional validations, such as multiplicity constraints are respected (which are checked by the above-mentioned EMF validation mechanism; these can be triggered in the generated EMF editor, or enabled to be performed automatically using "Live Validation"). For this reason, it is crucial to explicitly check that in the unit tests. In Appendix A, we will also implement a custom validator with a custom constraint check.

It is also crucial to create input models in these unit tests that give us enough guarantees that the migration is correct. In more complex scenarios, test input models with special corner cases should be carefully crafted.

3 Edelta: a versatile migration management

In this section, we demonstrate the versatility of Edelta by presenting four different application scenarios. In particular, we illustrate how the Java migrator, built on top of Edelta, can be employed to migrate models in various ways, depending on the requirements of application developers and clients. Specifically, we describe three approaches (in the following subsections) to migrating models within an EMF Eclipse application and discuss how to test the corresponding UI migration features. In Sect. 4.2, we will discuss and summarize the advantages and disadvantages of each approach.

In addition to the running example introduced in the previous section, we introduce a second, more complex example involving mutually dependent metamodels and models. This example highlights how Edelta can handle migrations in scenarios with intricate dependencies. Finally, we show how the same Java migrators of the two examples can be integrated into a REST service, demonstrating its applicability in contexts beyond Eclipse-based applications.

[6] This information was not available in the previous version [17].

[7] This has been added in this new version of Edelta.

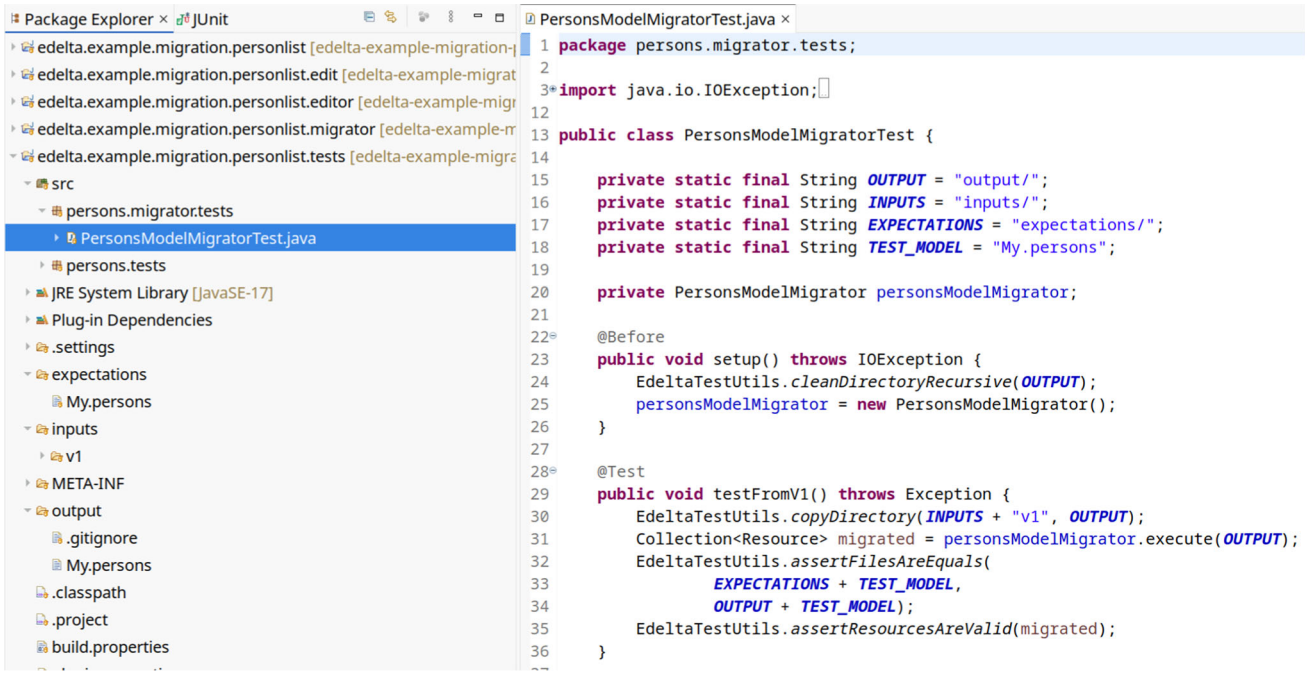


Fig. 6 A JUnit test for the migrator we have implemented

3.1 On editor opening migration

We must decide when to run the implemented Java migrator in our application. A possible strategy to execute the implemented Java migrator is when the EMF editor is opened on a model file. This way, the model file about to be edited will be automatically migrated if it needs to. Of course, since we are implementing an EMF application, we assume we already know how to customize the EMF-generated code.

We use `PersonsModelMigrator` when the EMF editor is opened on a model file before the editor loads the model into memory. The generated EMF editor loads the model into memory[8] by calling the method `createModel`. We customize such a method to first call our `PersonsModelMigrator` and then continue the execution: the original EMF-generated method will transparently use the migrated version of the model. Following the documented EMF strategy to customize generated code (and avoid further generation overwriting our customizations), we rename the generated method to `createModelGen`, leaving the `@generated` annotation, and implement our custom version of `createModel`. The editor exploits the `PersonsModelMigrator` by executing it on all the models in the current project (retrieved through the standard Eclipse API). The customized `createModel` method is shown in Listing 4. In this code, we stress that the "hardest" part is using the Eclipse API to retrieve the project's path of the opened model file. We kept the example minimal and simple.

Listing 4 The customized `createModel` method that first migrates the models in the current project.

```
/**
 * This is the method called to load a resource into the editing domain's resource set
 * based on the editor's input.
 * @generated
 */
public void createModelGen() { ... }

public void createModel() {
  PersonsModelMigrator migrator = new PersonsModelMigrator();
  // exception handling not shown
  migrator.execute(
    ResourceUtil.getFile(getEditorInput())
      .getProject().getLocation().toFile().getAbsolutePath());
  createModelGen();
}
```

Now, when clients open our EMF editor on an existing model, they will transparently get their models migrated (if needed) and made editable (Fig. 7). Note that the other model in the project has also been migrated (opened in the text editor in the figure).

Using the Eclipse API, as shown in Listing 5, we can further enhance the code to show a pop-up dialog with some information about the migration and its outcome.

[8] Technically, it loads the XMI model into the editor `EditingDomain`'s `ResourceSet`.

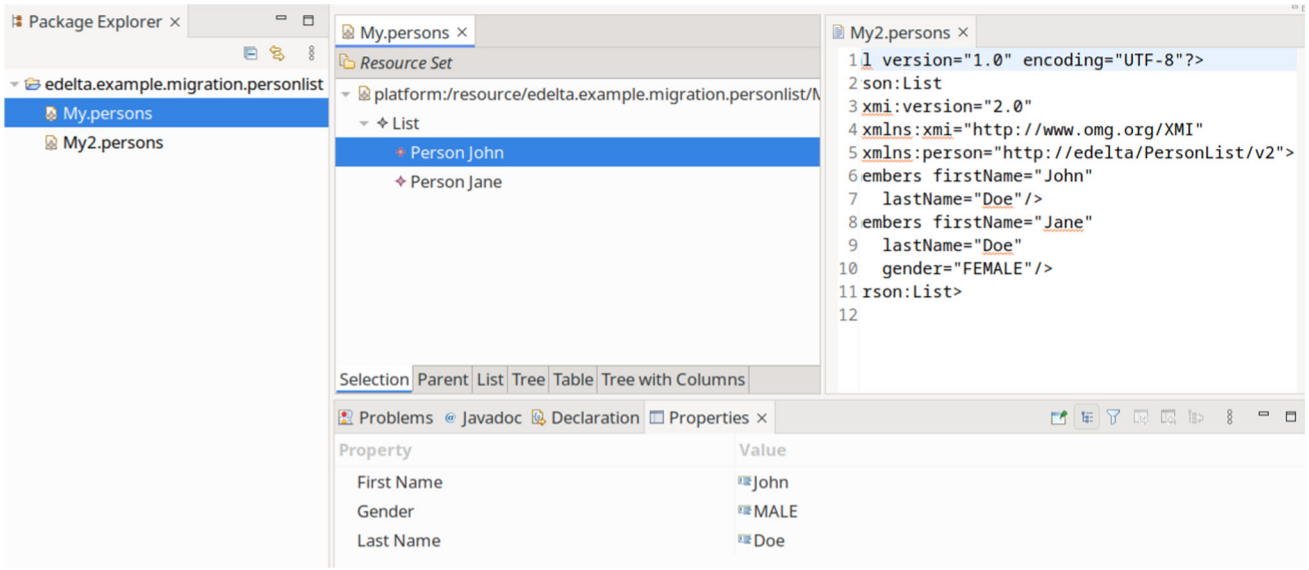


Fig. 7 The EMF editor can open the client's model because it first migrated the model

Listing 5 The improved `createModel` method that first migrates the models in the current project and shows a dialog with the migration outcome.

```
public void createModel() {
  PersonsModelMigrator migrator = new PersonsModelMigrator();
  String projectPath = ResourceUtil.getFile(getEditorInput())
    .getProject().getLocation().toFile().getAbsolutePath();
  // exception handling not shown
  Collection<Resource> migrated = migrator.execute(projectPath);
  if (!migrated.isEmpty()) {
    // show the migrated models in an information dialog
    MessageDialog.openInformation(
      getSite().getShell(),
      "Migration",
      "The following models have been migrated:\n\n" +
        migrated.stream()
          .map(m -> EdeltaResourceUtils.getRelativePath(m, projectPath))
          .collect(Collectors.joining("\n")));
  }
  createModelGen();
}
```

3.2 On-demand migration

While the migration on editor opening provides a seamless experience, there are scenarios where users or developers may want to explicitly trigger the migration of all models in the workspace. For example, after importing legacy projects or when batch-updating several models, an on-demand migration mechanism is desirable.

To support this, we implement a command in the Eclipse UI that allows users to invoke migration on all projects in the workspace. This command can be triggered from a menu entry or a toolbar button, allowing users to initiate the migration process at their convenience. The command's handler is implemented as shown in Listing 6 (the use of our migrator is similar to the one in Sect. 3.1).

Listing 6 The command handler that migrates all models in all projects of the workspace.

```
public class PersonsModelMigrationHandler extends AbstractHandler {
  @Override
  public Object execute(ExecutionEvent event) throws ExecutionException {
    IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
    IProject[] allProjects = root.getProjects();
    List<String> migratedResources = new ArrayList<>();
    // exception handling not shown
    for (IProject project : allProjects) {
      var projectPath = project.getLocation().toString();
      var personsMigrator = new PersonsModelMigrator();
      Collection<Resource> migrated = personsMigrator.execute(projectPath);
      migratedResources.addAll(migrated.stream()
        .map(m -> EdeltaResourceUtils.getRelativePath(m, projectPath))
        .toList());
    }
    IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindow(event
      );
    if (migratedResources.isEmpty()) {
      MessageDialog.openInformation(window.getShell(), "Migration",
        "No model files to migrate");
    } else {
      MessageDialog.openInformation(window.getShell(), "Migration",
        "The following models have been migrated:\n\n" +
          migratedResources.stream()
            .collect(Collectors.joining("\n")));
    }
    return null;
  }
}
```

The handler iterates over all projects in the workspace, invoking the `PersonsModelMigrator` on each. Any migrated resources are collected and reported to the user in an information dialog. If no migration is needed, the user is informed accordingly. Figure 8 shows the toolbar button (with the Eclipse icon) and the menu entry related to the handler on the left, and the dialog that appears after selecting the menu or the toolbar button after migrating models of two different projects on the right.

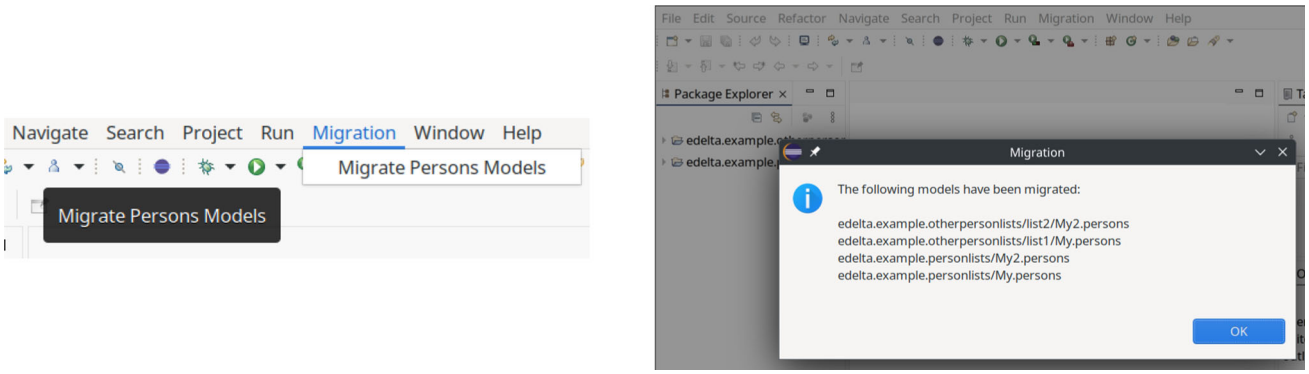


Fig. 8 The toolbar button (with tooltip) and the menu entry (left) and the dialog showing the migrated models in two different projects

Note that in both approaches, for simplicity, we only consider possible dependencies between models in the same project, i.e., intra-project dependencies. Both approaches can be extended to handle dependencies between models in different projects. In extra-project dependencies, both models must be migrated together (technically, in the same EMF `ResourceSet`), otherwise the standard EMF resolution mechanism might fail due to dangling references, leading to a failure in the migration process. A possible solution would be to migrate all the models of all the projects in the workspace, by passing all projects' paths together in a single call to `loadModelsFromPaths` (Sect. 2.2.1), which would handle all the models of all the projects together (i.e., in the same `ResourceSet`). However, dependencies among projects should be explicitly defined in clients' projects, e.g., through project dependencies in Eclipse. In particular, Eclipse would detect missing required projects in the workspace, and clients would be informed beforehand. In that respect, project dependencies are orthogonal to our approach and would not be interesting for the purposes of our work.

3.2.1 UI tests

To thoroughly validate the migration features from the user's perspective, we developed UI tests using SWTBot,[9] a widely adopted functional testing framework for Eclipse-based applications. SWTBot provides an API for the automation of user interactions with the Eclipse UI, such as clicking menu items, toolbar buttons, and interacting with dialogs and editors. Our SWTBot tests simulate real user interactions with the Eclipse UI, ensuring that both the on-demand and automatic migration mechanisms work as intended and that users receive the expected feedback. We create these tests in a brand new project "ui.tests".

Listing 7 shows an excerpt of the test class for the on-demand migration feature, which can be triggered either from the menu or the toolbar. These tests use some utility methods from the "Edelta UI Test Utils" feature for common setup operations, such as cleaning the workspace and importing the required test projects. Thanks to the readable API of SWTBot and the fluent API of AssertJ (the `assertThat` method), the tests should be easy to read and understand. The `testMenu` method checks that invoking the migration from the menu correctly migrates all relevant models in the workspace and that the user is informed about the migrated files. It also verifies that re-invoking the migration when no further models need migration results in an appropriate message. The `testToolbarButton` method ensures that the toolbar button triggers the same migration logic.

These tests are expected to be much slower than the unit tests, as they require a running Eclipse instance and the SWTBot framework to interact with the UI.

In addition to on-demand migration, we also test the on editor opening migration that occurs when a model is opened in the editor. Listing 8 demonstrates how the test verifies that only the models in the currently opened project are migrated, and that the editor can successfully display the migrated model.

These UI tests provide confidence that both migration strategies are correctly integrated into the user interface, that users receive clear feedback about migration actions, and that the migrated models are immediately usable in the editor. By automating these scenarios, we reduce the risk of regressions and ensure a smooth user experience as the migration infrastructure evolves.

Note that the UI tests are not meant to check that the migration logic is correct, as we already have unit tests for that (Sect. 2.2.2).

Finally, unless we change the strings of the dialog, the toolbar button, and menu entries, these tests will not need to be updated when we add new migrations or change the migration logic. In particular, we will always use the same test projects (containing input models of the first version of the Ecore), which will be updated with the latest version of

[9] https://projects.eclipse.org/projects/technology.swtbot

Listing 7 The UI test class for the on-demand migration feature.

```
@RunWith(SWTBotJunit4ClassRunner.class)
public class PersonsModelMigratorUiTest {

  private static SWTWorkbenchBot bot;

  @BeforeClass
  public static void beforeClass() {
    bot = new SWTWorkbenchBot();
    EdeltaUiTestUtils.closeWelcomePage();
    EdeltaUiTestUtils.openProjectExplorer();
  }

  @Before
  public void importProjects() throws Exception {
    EdeltaUiTestUtils.cleanWorkspace();
    // Import the projects needed for the tests
    EdeltaUiTestUtils.importProject("test-projects/edelta.example.
      personlists");
    EdeltaUiTestUtils.importProject("test-projects/edelta.example.
      otherpersonlists");
  }

  @Test
  public void testMenu() throws Exception {
    bot.shell().menu().menu("Migration").menu("Migrate Persons
      Models").click();
    SWTBotShell dialog = bot.shell("Migration").activate();
    // Verify that the dialog contains information that all workspace models have
      been migrated
    String dialogText = dialog.bot().label(1).getText();
    assertThat(dialogText)
      .contains(
        "The following models have been migrated:",
        "edelta.example.personlists/My.persons",
        "edelta.example.personlists/My2.persons",
        "edelta.example.otherpersonlists/list1/My.persons"
      ,
        "edelta.example.otherpersonlists/list2/My2.persons
      ");
    dialog.bot().button("OK").click();
    // Run migration again
    bot.shell().menu().menu("Migration").menu("Migrate Persons
      Models").click();
    // a subsequent click on the menu should show the dialog again
    // stating that no models have been migrated
    dialog = bot.shell("Migration").activate();
    dialogText = dialog.bot().label(1).getText();
    assertThat(dialogText).contains("No model files to migrate");
    dialog.bot().button("OK").click();
  }

  @Test
  public void testToolbarButton() throws Exception {
    bot.toolbarButtonWithTooltip("Migrate Persons Models").click();
    // ... as above
  }
}
```

Listing 8 The UI test method for the on editor opening migration feature.

```
@Test
public void testEditor() {
  SWTBotView projectExplorer = bot.viewByTitle("Project Explorer");
  SWTBotTreeItem testProject = projectExplorer.bot().tree().getTreeItem("
      edelta.example.personlists");
  testProject.expand().getNode("My.persons").select().doubleClick();
  SWTBotShell dialog = bot.shell("Migration").activate();
  // Verify that the dialog contains information that only the project models have
      been migrated
  String dialogText = dialog.bot().label(1).getText();
  assertThat(dialogText)
    .contains(
      "The following models have been migrated:",
      "edelta.example.personlists/My.persons",
      "edelta.example.personlists/My2.persons")
    .doesNotContain(
      "edelta.example.otherpersonlists/list1/My.persons",
      "edelta.example.otherpersonlists/list2/My2.persons")
      ;
  dialog.bot().button("OK").click();
  // Ensure that the EMF editor shows the (migrated model) tree
  bot.activeEditor().bot().tree().expandNode("platform:/resource/edelta
      .example.personlists/My.persons");
}
```

the Ecore and the corresponding model files, even in the subsequent migrations. As expected, these UI/integration tests are a bit harder to write than the unit tests. However, once written, they will not need to be updated, as said above.

3.2.2 A subsequent refactoring

Let us now evolve our application's metamodel again. We will use the Edelta refactorings library for the second refactoring, described in [15] and [16] concerning metamodels and models, respectively.

The Edelta program from version 2 to version 3 is shown in Fig. 9. As said above, the outline will show on-the-fly how the evolved metamodel will be (note the boldface elements representing modified/new elements). In this refactoring, the `enumToSubclasses` (see also [22]) replaces an enumeration attribute of a class by subclasses for each literal. In the code above, we refer to the attribute `gender`, whose literals are `MALE` and `FEMALE` (see Fig. 2). After the refactoring, `Person` will be turned into an "abstract" class, where the `gender` will be removed, and two new subclasses will be introduced, `Male` and `Female`. The implementation of this refactoring in the Edelta refactorings library also automatically handles the operations for adapting the models to such changes: an old instance of a `Person` with `gender=MALE` will be turned into an instance of the new `Male` subclass and similarly for `gender=FEMALE`.

Remember that for each new Edelta refactoring, we repeat the manual steps to regenerate the EMF Java code as shown in Sect. 2.1.

Since we have already implemented the main logic in the `PersonsModelMigrator` and in the EMF editor, we only need to modify `PersonsModelMigrator` by adding a call to `registerMigration` with the new Edelta program reference. In particular, neither the EMF editor nor the command handler needs to be modified. Thus, we only need to add a new line in `PersonsModelMigrator` as shown in Listing 9.

In the updated migrator, we also highlight that the order of registered migrations does not matter: the Edelta DSL compiler-generated Java classes contain all the needed information that allows the Edelta migration runtime to know from which version to which another version a model needs to be

Fig. 9 The second Ecore evolution with Edelta

Listing 9 The updated `PersonsModelMigrator` with the new migration registered.

```
1  public class PersonsModelMigrator {
2    public Collection<Resource> execute(String modelPath) throws Exception {
3  ...
4      migrator.registerMigration(FromV2ToV3::new); // newly added line
5      migrator.registerMigration(FromV1ToV2::new);
6  ...
```

migrated. The migrations, when needed, will be applied one after the other, in the correct order, until the models reach the state where they are compliant with the latest version of the metamodel.

After every new refactoring, we update the unit tests we saw above, ensuring the tests succeed. Of course, we must update the "expectations", which always represent a model instance of the current version of the metamodel.

Moreover, the SWTBot tests of Sect. 3.2.1 will not need to be updated, as they will still refer to the same input models and verify that the toolbar button, the menu, and the editor handle the migration to the latest version.

Then, we can manually verify that the client's models are updating by restarting the "client instance", and opening one of the model files. We could also verify that the current version of the client's model does not matter. However, this use case is already verified in the SWTBot tests, which will run all the migrations from the first version to the latest one.

The source code of this example, https://github.com/LorenzoBettini/edelta-example-migration-personlist,[10] also implements a third refactoring, which is not shown here for brevity. This is based on our refactoring library function `mergeAttributes`, which merges two attributes of a class into a single one.

3.3 On workspace startup migration

In this section, we focus on a scenario that is intentionally designed to highlight Edelta's ability to handle Ecore metamodels and model files with (possibly mutual) dependencies. The example's source code illustrated in this section can be found online at https://github.com/LorenzoBettini/edelta-example-migration-library.[11]

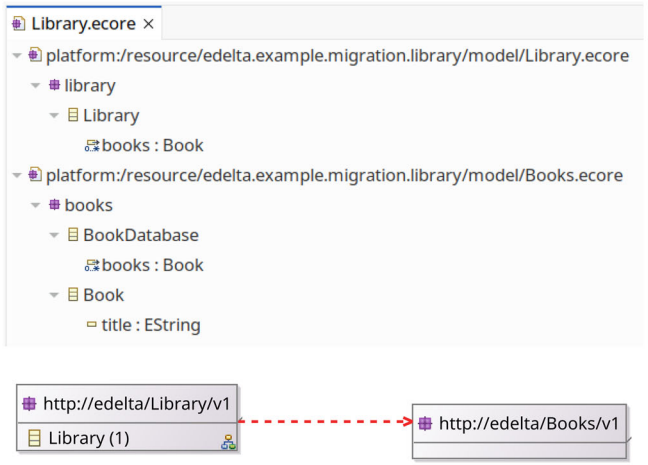
While mutual dependencies among metamodels might be considered a poor design choice—as they can lead to increased complexity and maintenance challenges—such situations may arise in practice, especially in legacy systems or when integrating independently developed metamodels. In any case, the purpose of this example is not to advocate for mutual dependencies but rather to demonstrate that Edelta is powerful and versatile enough to support migration tasks even in these less-than-ideal circumstances.

Moreover, although all classes could be defined in a single Ecore file and all model objects could be stored in a single file, it is often desirable to keep separate files for modularity or organizational reasons. The EMF tooling, for instance, allows the creation of separate model files with different root objects, even when using a single Ecore file. This flexibility is important for real-world scenarios, where metamodels and models are often split across several files, and references may cross file boundaries in complex ways. Thus, in the following, we will also use several input model files with mutual dependencies.

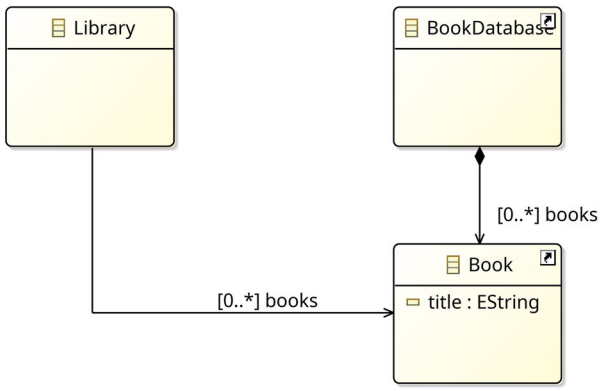
Figure 10 shows the initial versions of the two Ecore files. The first image depicts the Ecore files themselves, the second shows the dependencies between the packages (currently not mutual), and the third presents the corresponding class diagram. These initial metamodels define the structure of simple libraries and book databases, which will be the basis for the subsequent migrations.

[10] The source code of the example can also be found on Zenodo at https://doi.org/10.5281/zenodo.17294752.

[11] The source code of the example can also be found on Zenodo at https://doi.org/10.5281/zenodo.17597408.



(a) The initial Ecore files and package dependencies.



(b) The initial class diagram.

Fig. 10 The initial Ecore files, the dependencies, and the class diagram

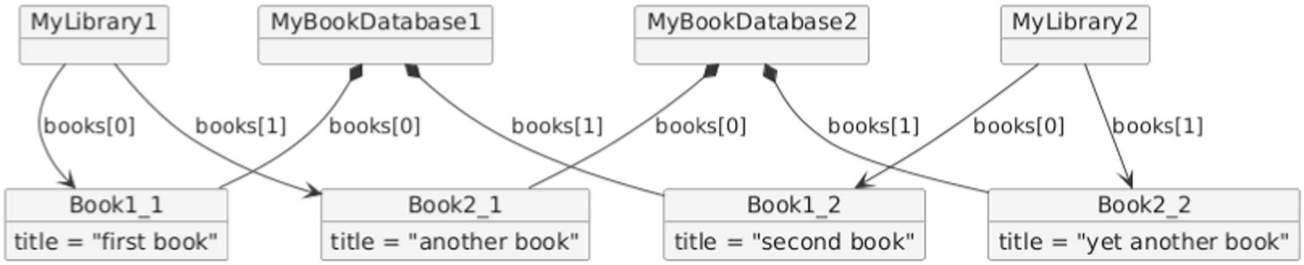


Fig. 11 The four initial models for the Library and BookDatabase Ecore files

To validate the migration process, we use four interconnected models as test cases: two conforming to the `Library` metamodel and two to the `Books` metamodel. These models are shown in Fig. 11. They serve as representative examples for both automatic and manual migration tests, ensuring that the migration logic correctly handles references and dependencies across multiple files.

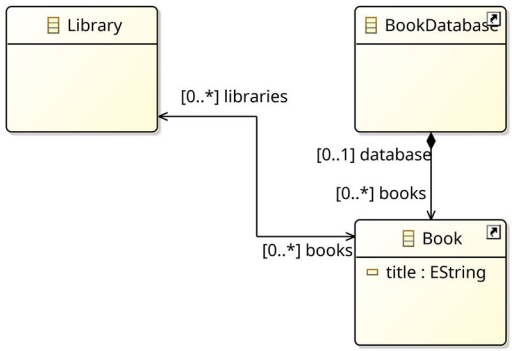
Following the same approach as in the previous example, we create a new "migrator" project, which contains the Edelta program to migrate the initial version of the two metamodels. For example, we want to introduce the missing opposite back references. This time, we must migrate two metamodels together. Thus, we copy both Ecore files into a subdirectory of `ecoreversions`. Also in this example, shown in Listing 10, we will use `v1`, `v2`, etc.

We use the fluent API of the Edelta runtime library to specify the metamodel changes. The Edelta DSL provides syntactic features that allow us to express the migration in a concise and readable way: the "with" operator `=>`, lambda expressions `[ ... ]`, and "extension methods" (allowing to call methods on the first argument as if the method were defined on the first argument's type). Note the use of fully qualified names for the reference `books` to avoid ambiguities (both metamodels have a reference named `books`). Recall that the Edelta editor provides content assist, so the developer can easily select the right reference to use. The migrated metamodels and the now mutual dependencies are shown in Fig. 12.

Listing 10 The Edelta program to migrate the two Ecore files from version 1 to version 2.

```
migrate "http://edelta/Library/v1" to "http://edelta/Library
    /v2"
migrate "http://edelta/Books/v1" to "http://edelta/Books/v2"

modifyEcore introduceOpposites epackage books {
  ecoreref(Book) => [
    // add a new EReference with name "database" and type BookDatabase
    addNewEReference("database", ecoreref(BookDatabase)) [
      // and make it bidirectional with the reference "books" in BookDatabase
      makeBidirectional(ecoreref(books.BookDatabase.books))
    ]
    addNewEReference("libraries", ecoreref(Library)) [
      makeMultiple
    ]
  ]
  ecoreref(library.Library.books).makeBidirectional(ecoreref(libraries))
}
```



(a) The class diagram.

(b) The mutual package dependencies.

Fig. 12 The diagram and dependencies after the first migration

While the EMF API and the EMF editors automatically set the back reference, this does not happen for existing models, which would not have the back references set. Instead, with a proper Edelta migration, models are updated automatically, and the back references are set correctly during the model migration performed by Edelta. In fact, the Edelta API used in the previous Edelta program automatically sets the back references when the model migration is performed.

Listing 11 The `LibraryModelMigrator` class that migrates models of both metamodels.

```
1  public class LibraryModelMigrator {
2    public Collection<Resource> execute(String modelPath) throws Exception {
3      EdeltaVersionMigrator edeltaMigrator = new EdeltaVersionMigrator();
4      edeltaMigrator.loadCurrentEPackages(
5        LibraryPackage.eINSTANCE, BooksPackage.eINSTANCE);
6      edeltaMigrator.registerMigration(FromV1ToV2::new);
7      edeltaMigrator.addModelFileExtensions(".library", ".books");
8      edeltaMigrator.loadModelsFrom(modelPath);
9      return edeltaMigrator.execute();
10   }
11 }
```

The implementation of the migrator class, shown in Listing 11, is similar to the previous example, but this time we have two metamodels in place. As in the previous example, we follow the approach of immediately testing that the migration is correct with a unit test. In particular, by comparing the input models and the output models after the migration, we can verify that the back references are correctly set (Fig. 13). The four models after the migration are shown in Fig. 14.

Instead of modifying the editor for running the migrator or providing commands for migration, in this example, we implement an Eclipse startup handler that executes automatically when Eclipse starts. Our handler will automatically scan and migrate models in the current projects of the workspace. Moreover, it will also register a project listener that will perform the migration operation on the projects that are possibly later opened or imported into the workspace.

Listing 12 The `plugin.xml` file registering the startup extension.

```
1 <plugin>
2   <extension point="org.eclipse.ui.startup">
3     <startup
4       class="edelta.example.migration.library.migrator.ui.
5         LibraryMigratorHandler">
6     </startup>
7   </extension>
</plugin>
```

We create a new project "migrator.ui" that contains the code to run the migrator. An extension point must be defined in the `plugin.xml` file of the `migrator.ui` project (Listing 12). Plug-ins that register a startup extension will be activated after the workbench initializes and have an opportunity to run code that cannot be implemented using the normal contribution mechanisms.

The code of the handler, shown in Listing 13, must be implemented only once, and it will be used for all the subsequent migrations of the Library and Books metamodels. As before, the code is intentionally simplified, hiding exception handling and other technicalities of the Eclipse API.

Now, if the workspace is opened with existing projects, the migrator will automatically check whether the models in the projects need to be migrated and will migrate them if needed. Moreover, we have registered a listener that will be notified when new projects are opened or imported into the workspace. In these new projects, the migrator will automatically check whether they need to be migrated and will migrate them if needed.

As in the previous example, we can test the migration using SWTBot. We create a new SWTBot project, which tests the migration of the models of the projects in the workspace. In Listing 14, we only show the relevant test (refer to the examples shown in Sect. 3.2.1).

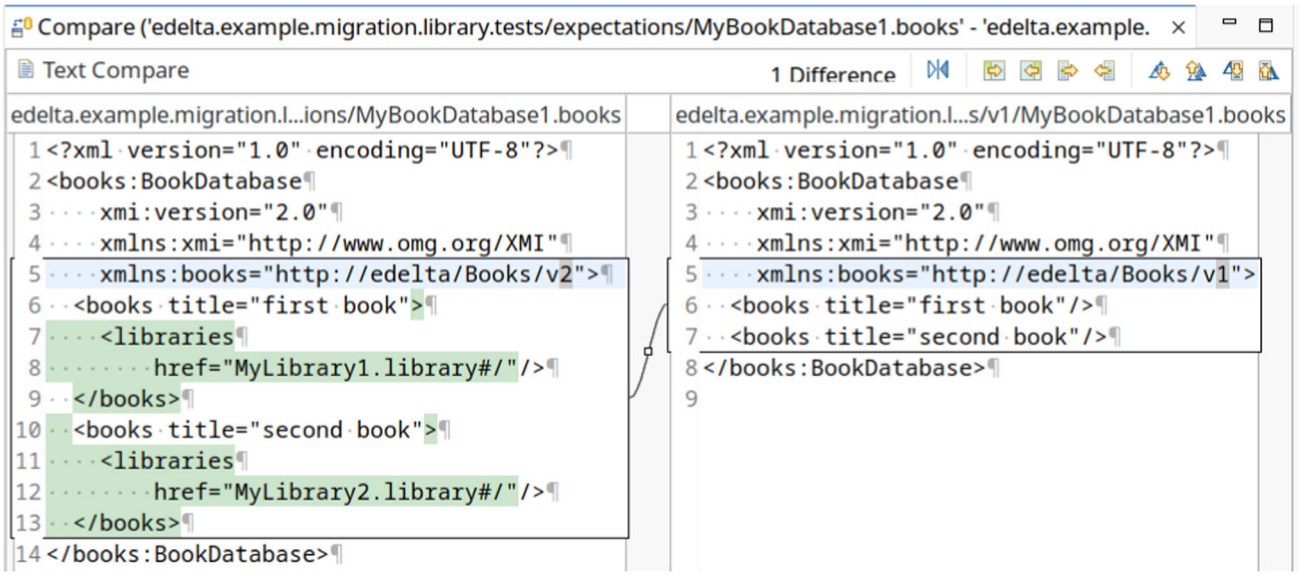


Fig. 13 One of the input models (right) and the migrated one (left) compared side by side

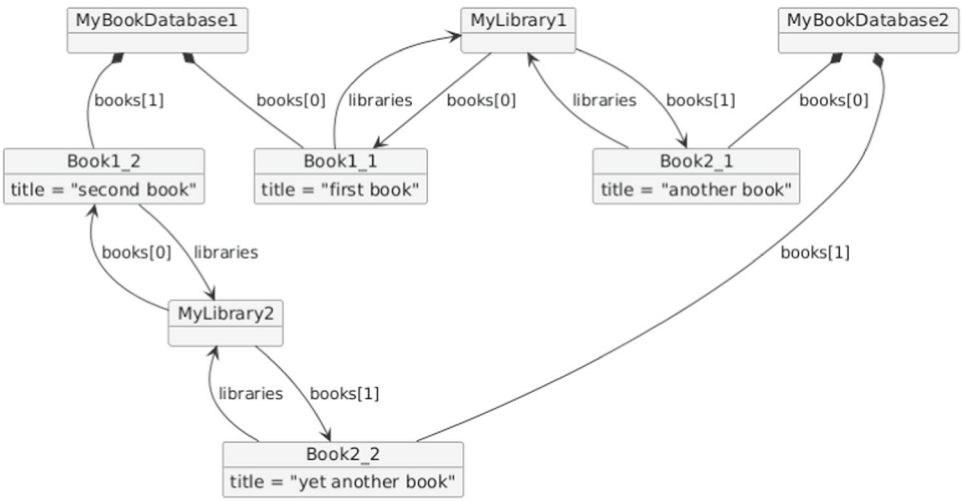


Fig. 14 The four models after the first migration

3.3.1 Another refactoring

The generated editor, due to the non-containment reference of `Library.books`, does not permit adding book references as children in the tree: book references can only be added in the properties view of the `Library` element, as shown in Fig. 15. It is not very user-friendly, so we want to change the metamodel to allow book references to be added as children of the library in the tree.

As shown in Listing 15, we apply the refactoring `referenceToClass`, which is part of the Edelta refactoring library, to the `Library` metamodel. This refactoring is useful when you want to represent additional information about the relationship (e.g., the number of book copies in a library, as we will do later) or allow the referenced objects to appear as children in the editor tree. In the context of this example, the reference `Library.books` is replaced by a containment reference to a new intermediate class, `BookItem`. We also rename the references to more appropriate names (`bookItems` and `libraryBookItems`) to avoid confusion.

This refactoring of the Edelta library not only updates the metamodel structure but also takes care of migrating existing models: for each original reference, a corresponding `BookItem` instance is created, and the reference to the `Book` is preserved. This ensures that the semantics of the original models are preserved and that the new structure is immediately usable in generated editors and tools. For further technical details and the implementation of this refactoring, see the Edelta GitHub repository and [16].

As usual, we run the Edelta program, copy the modified Ecore files in the model project, reload the GenModel, and

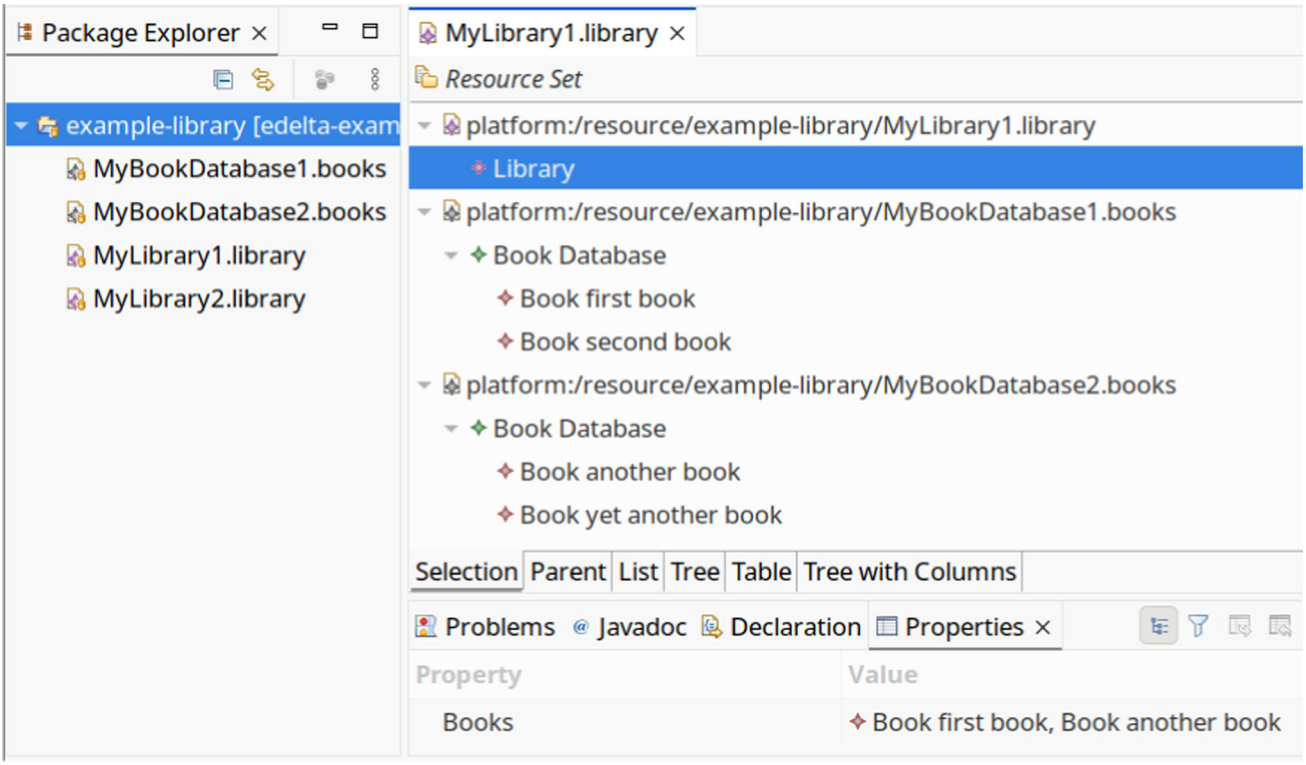
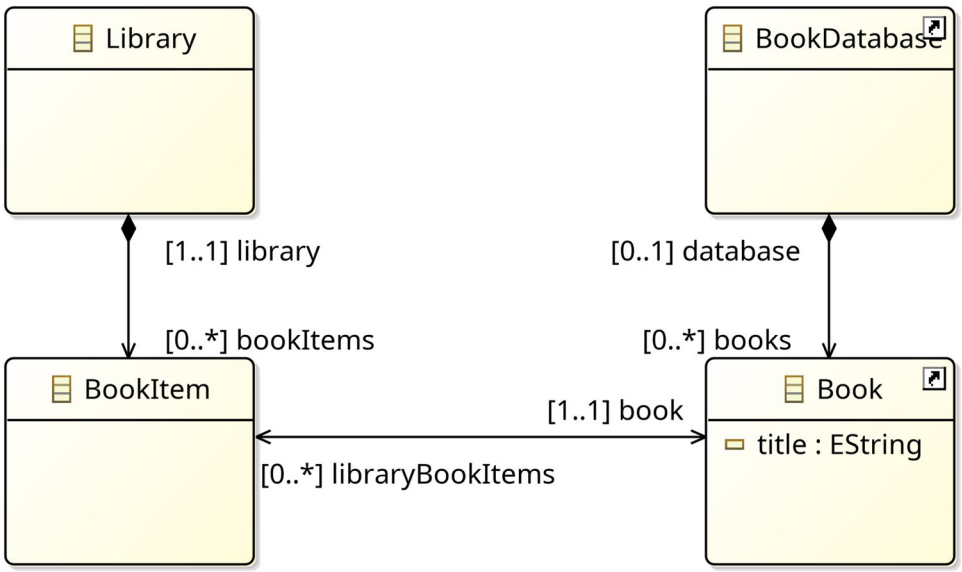


Fig. 15 The initial editor for the Library Ecore

Fig. 16 The diagram after the second migration



regenerate the code. The resulting class diagram is shown in Fig. 16.

As before, we update the migrator to register this new migration. Moreover, we add another unit test and update the expectations.

Now the EMF editor allows adding book references as children of the library in the shape of `BookItems`, as shown in Fig. 17. Moreover, we can add several `BookItem` objects referring to the same `Book` object. A library can have several copies of the same book, which we want, and was not possible before this refactoring.

Note that the applied refactoring introduces an additional level of indirection in the model structure: instead of directly referencing `Book` objects from a `Library`, we now have `BookItem` objects that reference `Book` objects. This change can have implications for how we interact with the

Listing 13 The `LibraryMigratorHandler` class that automatically migrates models in the workspace at startup and when new projects are opened or imported.

```
public class LibraryMigratorHandler implements IStartup {
  @Override
  public void earlyStartup() {
    IWorkspace workspace = ResourcesPlugin.getWorkspace();
    // Process existing projects in the workspace
    IProject[] projects = workspace.getRoot().getProjects();
    processProjects(projects);
    // Register listener for new projects (opened/imported)
    workspace.addResourceChangeListener(new ProjectListener(),
      IResourceChangeEvent.POST_CHANGE);
  }

  private void processProjects(IProject... projects) {
    List<String> migratedResources = new ArrayList<>();
    for (IProject project : projects) {
      String projectPath = project.getLocation().toString();
      LibraryModelMigrator libraryMigrator = new LibraryModelMigrator();
      Collection<Resource> migrated = libraryMigrator.execute(projectPath);
      migratedResources.addAll(migrated.stream()
        .map(m -> EdeltaResourceUtils.getRelativePath(m, projectPath))
        .toList());
    }
    if (!migratedResources.isEmpty())
      MessageDialog.openInformation(window.getShell(), "Migration",
        "The following models have been migrated:\n\n" +
          migratedResources.stream()
          .collect(Collectors.joining("\n")));
  }

  private class ProjectListener implements IResourceChangeListener {
    @Override
    public void resourceChanged(IResourceChangeEvent event) {
      if (event.getType() == IResourceChangeEvent.POST_CHANGE) {
        event.getDelta().accept(delta -> {
          if (delta.getResource() instanceof IProject project) { // other checks not
    shown
            processProjects(project);
            return false; // don't visit children
          }
          return true; // keep visiting
        });
      }
    }
  }
}
```

Listing 14 The UI test method for the workspace startup migration feature.

```
@Test
public void testMigrationOnWorkspace() throws Exception {
  // import the first project
  EdeltaUiTestUtils.importProject("test-projects/firstproject");
  // verify the pop-up dialog shows the migrated models of the first project
  SWTBotShell dialog = bot.shell("Migration").activate();
  String dialogText = dialog.bot().label(1).getText();
  assertThat(dialogText)
    .contains(
      "The following models have been migrated:",
      "firstproject/MyBookDatabase1.books",
      "firstproject/MyLibrary1.library");
  dialog.bot().button("OK").click();
  // import the second project
  EdeltaUiTestUtils.importProject("test-projects/secondproject");
  // verify the pop-up dialog shows the migrated models of the second project
  dialog = bot.shell("Migration").activate();
  dialogText = dialog.bot().label(1).getText();
  assertThat(dialogText)
    .contains(
      "The following models have been migrated:",
      "secondproject/MyBookDatabase2.books",
      "secondproject/MyLibrary2.library")
    .doesNotContain("firstproject");
  dialog.bot().button("OK").click();
}
```

Listing 15 The Edelta program to migrate the two Ecore files from version 2 to version 3.

```
import edelta.refactorings.lib.EdeltaRefactorings

migrate "http://edelta/Library/v2" to "http://edelta/Library
    /v3"
migrate "http://edelta/Books/v2" to "http://edelta/Books/v3"

use EdeltaRefactorings as refactorings

modifyEcore applyReferenceToClass epackage library {
  refactorings.referenceToClass("BookItem", ecoreref(library.Library.books))
  ecoreref(Library.books).name = "bookItems"
  ecoreref(Book.libraries).name = "libraryBookItems"
}
```

models, especially when accessing or manipulating books within a library. Moreover, it could introduce some overhead due to the additional objects created. However, this structure is more flexible (we can refer to the same book multiple times and the usability of the editor is improved, as said above) and allows for richer representations of the relationships between libraries and books, especially when additional attributes or behaviors need to be associated with the relationship itself. In fact, in Appendix A, we show how to implement another refactoring that adds a new attribute to the `BookItem` class, which allows specifying the number of copies of a book in a library.

3.4 Embedding migrators in a REST service

This section describes how to realize a REST service for model migration using the Java migrators we implemented in the previous section. The REST service shows how the Java migrators, based on Edelta, can be applied in several contexts beyond Eclipse-based applications. The source code of this example is available at https://github.com/gssi/edelta-migrationservice.[12]

Figure 18 shows the API documentation of the implemented REST service. It takes a set of models to be migrated as input and returns the migrated models in a zip file. We implemented the REST service using the Spring Boot framework[13] and Apache Maven as a dependency management system.[14]

Listing 16 illustrates the Java method related to the POST method of the REST services (as before, exception handling and logging statements are not shown). The `createFolderWithInputModels` method stores the

[12] The source code of the example can also be found on Zenodo at https://doi.org/10.5281/zenodo.17598181.

[13] https://spring.io/projects/spring-boot

[14] https://maven.apache.org/

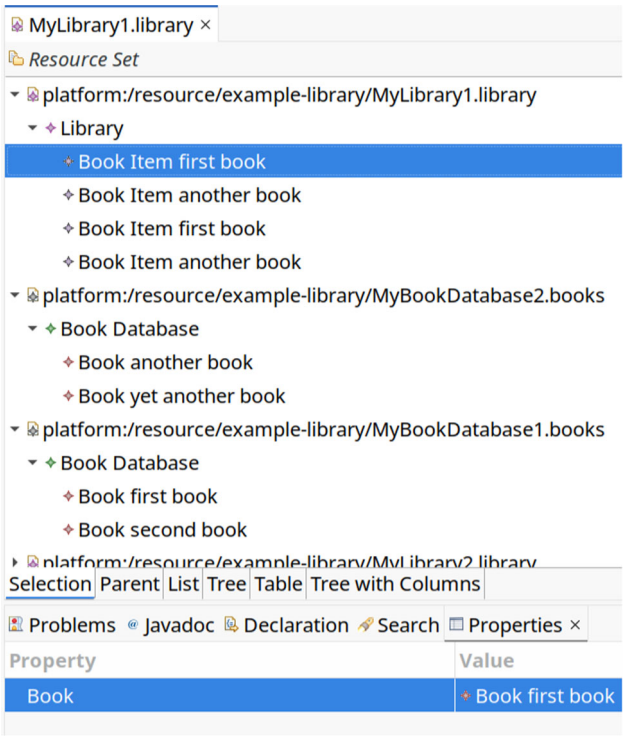


Fig. 17 The new editor for the Library Ecore

Listing 16 The REST service method that migrates the input models.

```
@PostMapping("/")
public ResponseEntity<byte[]> modelmigration(@RequestParam MultipartFile[]
        modelFiles) {
    Path newFolder = createFolderWithInputModels(modelFiles);
    Collection<Resource> result = new ArrayList<>();
    // first try with the PersonsModelMigrator
    PersonsModelMigrator personsModelMigrator = new PersonsModelMigrator();
    result.addAll(personsModelMigrator.execute(newFolder.toString()));
    // then try with the LibraryModelMigrator
    LibraryModelMigrator libraryModelMigrator = new LibraryModelMigrator();
    result.addAll(libraryModelMigrator.execute(newFolder.toString()));

    byte[] zipBytes = createZipFileResponse(result);
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.parseMediaType("application/zip"));
    headers.setContentDispositionFormData("attachment", "
        migratedmodels.zip");
    headers.setContentLength(zipBytes.length);
    FileSystemUtils.deleteRecursively(newFolder);
    return new ResponseEntity<>(zipBytes, headers, org.springframework.http.
        HttpStatus.OK);
}
```

received input models in a new temporary folder. Then, we create an object of the class `PersonsModelMigrator` defined in Sect. 2.2.1. The `execute` method is invoked to migrate the models related to the `Persons.ecore` metamodel stored in the previously created folder. Similarly, we then employ the `LibraryModelMigrator` (Sect. 3.3) class to migrate the models for the `Library` and `Books` metamodels. When other models related to a different metamodel should be migrated, we follow a similar approach. Since both migrators are used and the migrators only load and possibly migrate model files with the specified extensions, the two migrators can be used together without any issues. In particular, in the same REST POST request, we can migrate models related to both the `Persons.ecore` and the `Library.ecore` and `Books.ecore` metamodels.

Listing 17 The Maven dependencies to use the two migrators in a Java application.

```
<dependency>
  <groupId>io.github.lorenzobettini.edelta.examples</groupId>
  <artifactId>edelta.example.migration.personlist.migrator</artifactId>
  <version>1.0.0</version>
</dependency>
<dependency>
  <groupId>io.github.lorenzobettini.edelta.examples</groupId>
  <artifactId>edelta.example.migration.library.migrator</artifactId>
  <version>1.0.0</version>
</dependency>
```

The migrators of the two examples are deployed as Maven artifacts to the Maven Central repository, so using them in a Java application like this Spring Boot REST service is straightforward: it is enough to add the Maven dependencies to the `pom.xml` file of the project, as shown in Listing 17. Transitive dependencies are automatically resolved as usual in Maven.

As done in the rest of the paper, automated tests are crucial to ensure the REST service works correctly. Using Spring testing features, we can write a few integration tests with Spring MockMvc, which allows us to simulate HTTP requests and verify the responses without needing a running server. These tests are still written with JUnit by using the Spring annotation `@SpringBootTest`, which properly sets up the Spring application context for testing. Such tests, included in the test class `MigrationControllerTest`, can be found in the example's source code. The test suite validates the migration functionality by uploading multiple XMI model files as multipart form data to the REST endpoint, receiving a ZIP archive containing the migrated models in response. Each test case loads input models and expected output models from the test resources, performs the migration request, and then extracts and compares the contents of the returned ZIP file against the expected results using normalized string comparisons. The tests cover individual metamodel migrations (e.g., personlist and library metamodels) as well as combined scenarios where models from different metamodels are migrated simultaneously, ensuring that the service correctly handles both single-domain and cross-domain migration requests.

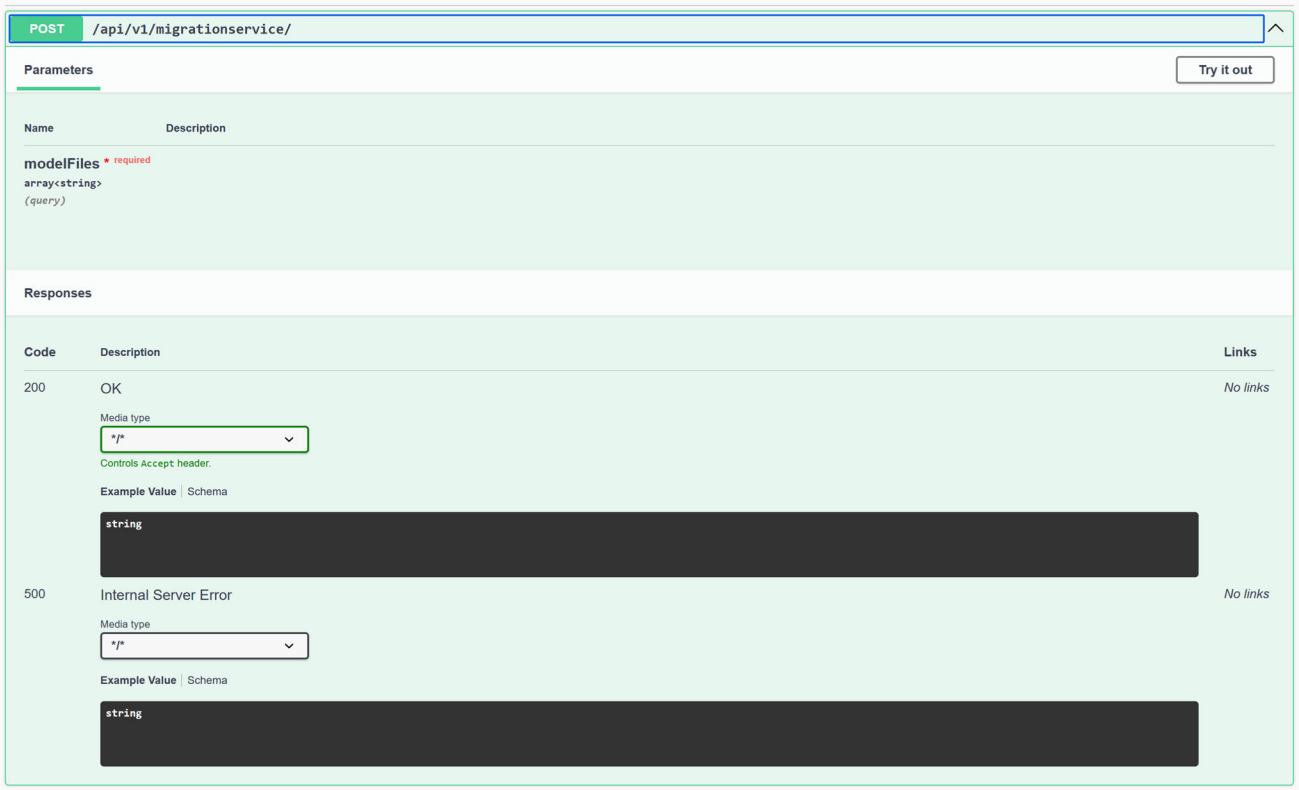


Fig. 18 Migration REST service

4 Discussion

In this section, we discuss our experience with Edelta and the migration of EMF models. First, we generally discuss the implementation of the migration logic, which is independent of the running platform, e.g., Eclipse. Then, we focus on the strategies for migrating models in an Eclipse application to keep them consistent with the latest metamodel version. We also discuss these strategies' implications (advantages and possible drawbacks/pitfalls) for developers and end-users. We also show some performance measurements we performed to assess the scalability and efficiency of our model migration framework. Finally, we briefly describe how we developed and tested Edelta itself, relying on automated tests and continuous integration.

4.1 The migration logic

Edelta can handle the migration of "stale" models from any past version to the latest one, performing the needed migrations one after the other using the `nsURI` information specified in the Edelta programs. A "lazy" client who waits a long time before migrating to the latest version of an application will not experience any problem when finally updating the application: the models will be migrated and updated. Depending on the strategy implemented in the Eclipse application, the migration can be performed automatically or on demand.

We showed how the Edelta DSL permits the concise definition of the evolution and refactoring of the metamodels. Moreover, the Edelta compiler and its integration with Eclipse statically checks most of the metamodel evolution and refactoring operations, avoiding possible incorrect evolved metamodels and providing immediate feedback to the developer, e.g., through the outline view, which shows the current state of the evolved metamodels.

Using the Java code generated by Edelta, developers can straightforwardly implement the migration logic. The evolution and logic of the migrations of the metamodels must be ensured to be correct by the EMF application developer. As we illustrated, the Edelta refactorings library can handle the evolutions of models in some circumstances, but in other ones, Edelta requires a minimal amount of code to consistently migrate the models (see [15, 16] and Appendix A for more details).

Developers should "validate" the evolutions and refactorings before deploying a new application version using automated tests. In particular, as we showed in Sects. 2 and 3, the correctness of migrations should be checked with unit tests, while the correctness of the migration usage in the UI

should be verified with UI tests. In that respect, in this paper, we also showed some new additional testing libraries provided by Edelta, which can be used to test the migrations and the UI integration.

To experiment outside of the Eclipse ecosystem, we also implemented a REST service that may act as a "migrator-as-a-service" [23], especially useful in MDE cloud-based environment [24, 25] and model-based repositories. This demonstrates that the Java code generated by the Edelta compiler can be used in any Java-based environment, even outside of Eclipse. In fact, the generated Java code only depends on the Edelta runtime library and the standard EMF API, which are both independent of Xtext and Eclipse. In particular, the migration logic of the two examples is being reused without any modification in the REST service. Again, this shows that the small effort required to implement the migration logic in Java is repaid by its usability in different contexts.

Moreover, using a REST service approach allows for greater scalability when migrating larger models, i.e., serving several requests from different clients. In particular, regarding vertical scalability, the REST service can be deployed using any cloud system by defining the required CPU, RAM, and other resources. In this way, the REST service can scale up to allow managing requests containing large models. For horizontal scalability, the REST service can be dockerized and utilized on cloud platforms that contain numerous virtual machines. This scaling out enables the management of the increasing number of clients requesting model migrations. Another advantage of using the Spring Boot framework to implement the REST service is that concurrent access is transparently managed by the framework, therefore, allowing several clients to access the REST service simultaneously to migrate models. Since each request is served by a different thread, and each request uses a separate temporary folder to store the models during the migration process, there are no concurrency issues when multiple clients access the REST service at the same time. The last interesting point is about security; specifically, the Spring Security framework[15] can be used to provide various authentication and access-control mechanisms.

4.2 Strategies for migrating models in Eclipse applications

As we said, it is up to the developers to decide the best moment to run the check and migration process. In this paper, we showed three different strategies for running the migration logic in Eclipse: "on editor opening" (Sect. 3.1), "on-demand" (Sect. 3.2), "on workspace startup" (Sect. 3.3). In the following, we summarize the main features of these strategies and their implications for developers and end-users.

We start with the advantages that are common to all the strategies we presented:

- *Minimal changes* All strategies require minimal code to add to generated EMF classes or to create in classes that extend the Eclipse user interface (also known as "extension points"). Moreover, such an amount of code has to be written only once, as the migration logic is decoupled from the Eclipse user interface, and the code related to the Eclipse user interface does not need to be updated even when implementing further metamodel migrations in the same application.
- *Extensibility* Since the migration logic and the user interface are decoupled from each other, they can be independently extended or customized as needed.
- *Clear user feedback* In all the approaches, providing immediate and comprehensive feedback about which models have been migrated or if no migration was necessary is straightforward. The Edelta API provides all the information needed to generate such feedback, including the list of models that were migrated.
- *Repeatability and safety* the migration can be executed at any time, and only models that require migration will be affected. In particular, using the migrator on already up-to-date models will not cause any issues, as the migration mechanisms implemented by Edelta are designed to be idempotent. Moreover, when models are already up to date, the migration process has a minimal overhead, as the migrator will simply check the model's metamodel version and to do that it does not need to perform heavier operations like resolving EMF cross references.

Let us now define a few desirable properties of the migration strategies, which we will use to compare the three approaches we presented.

- *Transparent user experience* Checks for models needing migration and the actual model migration are executed automatically. Users are not required to perform any manual migration steps, and the process is seamless from their perspective. This belongs to the first and third strategies (the migration occurs when a model file is opened in the editor for the first strategy, and when the workspace is started for the third strategy). On the contrary, the second strategy requires users to explicitly invoke a command to migrate models, which may not be as transparent.
- *Robustness to workspace changes and workspace-wide migration* Models should always be updated with the latest metamodel version before the user edits them. This belongs to the first and third strategies but not to the second one for the same reason. However, unlike the

[15] https://spring.io/projects/spring-security

first strategy, which is project-scoped, the second (when explicitly invoked) and third strategies migrate models across all projects in the workspace in a single operation, ensuring consistency after bulk operations such as project imports.

- *User-driven control* Migration is performed only when explicitly requested by the user, giving complete control over when models are migrated. This is particularly useful when users want to review, back up, or coordinate migrations with other project activities. This is a property of the second strategy, allowing users to invoke the migration command at their convenience. The first and third strategies do not provide this level of control, as they automatically migrate models when they are opened or when the workspace starts. In this respect, the second strategy is less intrusive, as it does not interfere with the user's workflow unless they explicitly invoke the migration command.

Table 1 provides a comparison of the three strategies. The symbols indicate whether a strategy fully supports (✓), partially supports (∼), or does not support (−) a property.

Let us now discuss the potential drawbacks of the strategies we presented.

Concerning the first strategy, a potential limitation is that only models in the currently opened project are migrated when the editor is invoked. Models in other projects will be migrated only when they are accessed. However, this delay does not impact the correctness of the models that the clients will edit, since as soon as an out-of-date model is opened in a different project, that project's models will also be migrated. Moreover, this strategy could also be easily extended to migrate models in all projects in the workspace when the editor is opened.

Concerning the second strategy, potential limitations include the need for users to remember to invoke the migration after adding new projects or models, and the possibility of models being left outdated if the migration is not triggered. However, this explicitness is often desirable in some settings, where users may want to coordinate migrations with other activities or maintain control over the timing of changes (e.g., users wish to review or back up models before migration). Moreover, users will realize the need to migrate models when they open an outdated model in the editor, as the editor will show several error messages. We can say that the on-demand migration complements the first strategy by providing flexibility, explicit control, and workspace-wide consistency.

A potential drawback of the third strategy is that users can disable the execution of specific extensions in their workspace preferences (e.g., via the Preferences: General, Startup and Shutdown, Plug-ins activated on startup). This, of course, would prevent the migrator from running.

The developer could also implement alternative strategies to run the migration logic in an Eclipse application, depending on the application's specific needs and its users. For example, the migration logic could be executed by an Eclipse project builder, which is a mechanism that allows developers to define custom build steps for their projects. Such a builder could be configured to run the migration logic whenever a project is built, ensuring that models are always up to date with the latest metamodel version. This would be similar to the third strategy, but it would allow for more fine-grained control over when the migration is executed, as specific build events or conditions could trigger it. However, implementing such a builder would require additional effort and complexity, which would be imposed only by the Eclipse building mechanism; the implemented migration logic could be reused without changes. In an Eclipse RCP application,[16] where there might not be a workspace, the strategies shown in this paper could still be reused. Of course, the strategies should be updated not to rely on projects in a workspace but on paths in the file system.

4.3 Performance considerations

To assess the scalability and efficiency of our model migration framework, we conducted a performance evaluation measuring execution time under varying workload conditions. We used the second example, which involves migrating multiple interconnected model files of two metamodels.

The performance tests were implemented as a standalone Java application[17]. Test models are generated programmatically based on the structure of the original v1 test models used in Sect. 3.3. We have two independent measurement series: *Number of files:* varies the number of input files from 4 to 1024 (doubling at each step) while keeping model size constant (2 books per database file and the corresponding references in the library file); *Model size:* varies the total number of model elements from 64 to 16,384 (doubling at each step) while keeping the number of files constant at 4. Execution time is measured using `System.nanoTime()` for high-resolution measurements. For each measurement point, inputs are regenerated before each iteration to ensure models start at version 1.

The size variation test duplicates all elements in both `.books` files (book elements) and `.library` files (book references), ensuring proportional growth across all model types. Library references are updated to point to the correct duplicated book indices, maintaining model consistency.

[16] The Eclipse Rich Client Platform permits implementing any client application, not necessarily based on an IDE.

[17] `EdeltaLibraryModelMigratorPerformanceStatistics` within the test module of the library example https://github.com/LorenzoBettini/edelta-example-migration-library

Table 1 Comparison of migration strategies and their properties

Property	On Editor Opening	On Demand	On Startup
Transparent user experience	✓	–	✓
Robustness to workspace changes	~	–	✓
Workspace-wide migration	–	✓	✓
User-driven control	–	✓	–

✓ = fully supported; ~ = partially supported; – = not supported

Table 2 Performance measurements

(a) Number of files

Files	Time (ms)
4	18.38
8	22.03
16	35.35
32	43.02
64	62.73
128	82.40
256	117.72
512	216.31
1024	350.00

(b) Model size

Total Elements	Time (ms)
64	4.08
128	4.69
256	13.35
512	17.89
1024	18.77
2048	30.84
4096	52.20
8192	102.38
16384	171.42

We conducted performance measurements on multiple platforms. The primary results were obtained on a Dell Precision Tower 7000 Series (Tower 7910) workstation with an Intel Xeon processor, 24 cores, and 32 GB of RAM, representing a typical high-performance development environment. Additional measurements were performed on a portable laptop (LG Gram) and on GitHub Actions runners (Linux, macOS, and Windows) to evaluate performance across different hardware and operating system configurations.

Table 2a and b present the detailed performance measurements from the Dell workstation. The results show that execution time scales roughly linearly with the number of files (Table 2a). Migrating 4 files takes approximately 18 ms, while 1024 files require 350 ms, representing a 19× increase in time for a 256× increase in file count. This sublinear scaling indicates efficient handling of multiple files, likely due to shared infrastructure and resource reuse during batch processing. The model size measurements (Table 2b) reveal interesting behavior. For small models (64–128 elements), execution time remains extremely low (4 ms) with minimal variation, suggesting that fixed overhead dominates for small inputs. As model size increases beyond 256 elements, execution time grows approximately linearly, reaching 171 ms for 16,384 elements. The small variations in the lower range are within measurement noise and likely reflect JVM optimization effects.

All the performance results can be found on the GitHub repository of the example project.[18] They are also visualized in Figs. 19 and 20, which plot execution time against the number of files and model size, respectively, in all the tested platforms. The figures use logarithmic scales on Y-axes to better visualize the growth behavior across the wide range of values.

Performance measurements on the LG Gram laptop show similar trends but with approximately 2–3× longer execution times, which is expected given the lower-power mobile processor. Notably, the scaling behavior remains consistent: file count and model size both exhibit near-linear growth patterns, confirming that the algorithmic complexity is platform-independent.

The GitHub Actions runners provide insight into cloud-based CI/CD environments:

- The *Linux runner* (Ubuntu 24.04) shows performance comparable to the LG Gram laptop, with 1024 files taking 2.09 s and 16,384 elements requiring 953 ms.
- The *macOS runner* (macOS 15.7.1 on ARM64) exhibits similar performance to Linux, though with slightly lower times for small models, possibly reflecting Apple Silicon's optimization characteristics.
- The *Windows runner* (Windows Server 2025) shows the slowest performance among GitHub Actions environments, with 1024 files requiring 3.66 s, approximately 1.7× slower than Linux, possibly reflecting the lower performance of the NTFS file system and the operating system in general.

[18] https://github.com/LorenzoBettini/edelta-example-migration-library

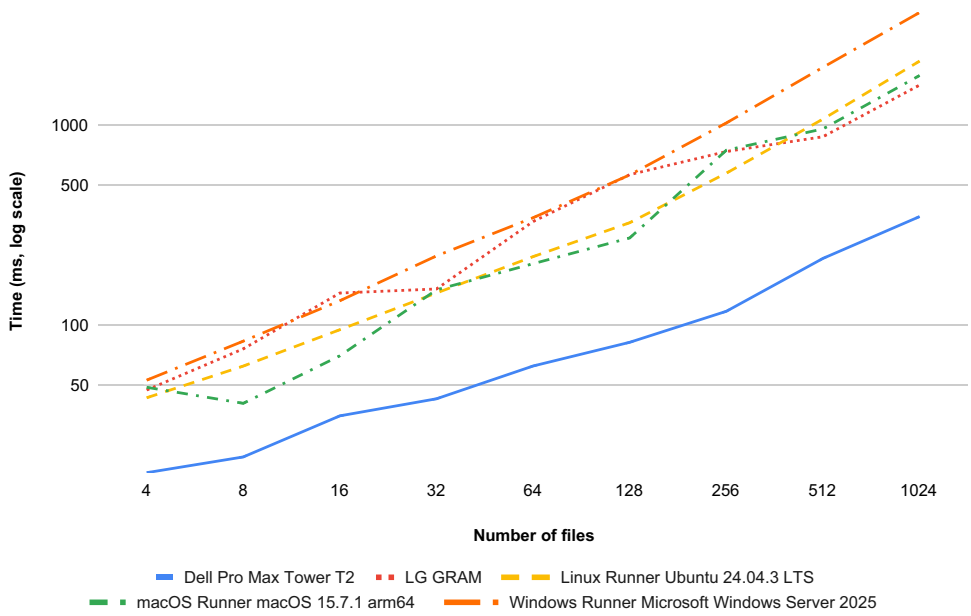


Fig. 19 Performance measurements plots (Number of files)

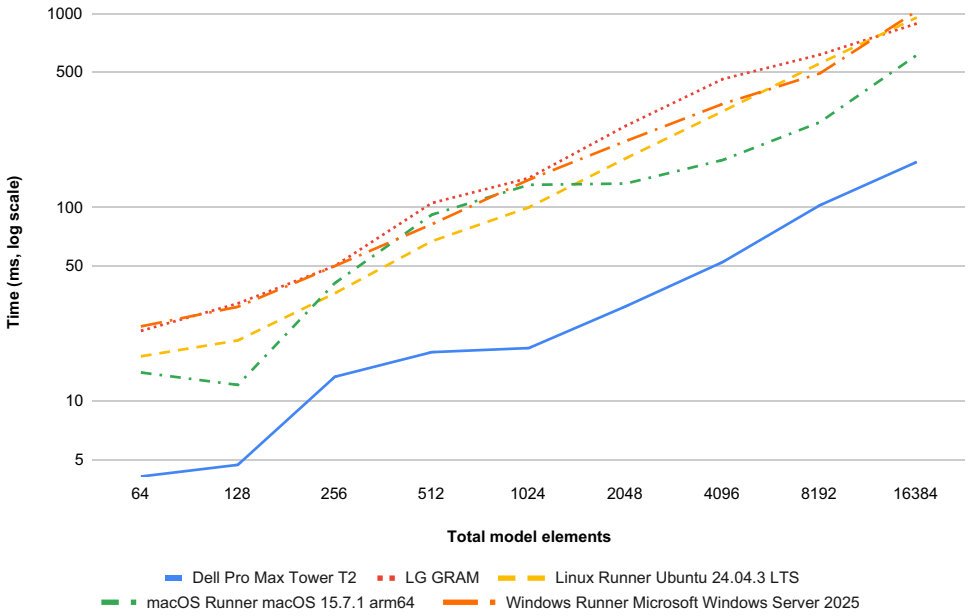


Fig. 20 Performance measurements plots (Model size)

If we exclude the performance on the Dell workstation, which is significantly better than the other platforms, due to its high-end hardware, we can see that the performance starts degrading more notably when the numbers tend to reach the limits of the underlying hardware and the operating system. This concerns both file count (I/O operations) and model size (memory usage and the corresponding garbage collection overhead). Moreover, with such huge numbers, Eclipse itself might not be able to handle such many files or large models efficiently, independently of our migration framework. This also holds for the EMF framework and its UI components (e.g., the tree viewer used in the model editor).

Despite these absolute time differences, all platforms maintain the same scaling characteristics for reasonably sized inputs (being typical model sizes and file counts used in practice), making the results encouraging for practical use across diverse environments. As a reference, in the ModelSet repository [26], which is one of the largest model repository project, the average model size is reported to be around 150 instances for UML models and 200 for Ecore models, while our trend starts to degrade to around 1500 instances. On modern development workstations, migrating typical model sizes (hundreds to thousands of elements) completes in tens to hundreds of milliseconds, which is negligible for interactive workflows. Even at the upper end of our test range (1024 files

with 16,384 total elements), execution completes in under 200 ms on the Dell workstation, well within acceptable limits for automated migration tasks. The consistency of results across platforms for reasonable inputs, including cloud-based CI/CD runners, confirms that our migration framework is also suitable for automated integration pipelines where performance predictability is crucial. Moreover, the impact on end-user experience in Eclipse applications is minimal, as migrations complete quickly enough to avoid noticeable delays during model loading or editing.

The `EdeltaVersionMigrator` also provides a method (`isMigrationNeeded`) to check if migration is needed without performing it. As soon as a model needing migration is found, the check immediately returns true. This method can be used to enhance the user experience in Eclipse applications. For example, in the first strategy (Sect. 3.1), when opening the editor, the application could first check if migration is needed and, before performing it, ask the user for confirmation or at least inform them about the need for migration and its implications (e.g., time required, backup creation, etc.). This could also be applied to the other two strategies. Note that such a change in the UI would require adapting the SWTBot UI tests accordingly. Listing 18 shows an alternative implementation of the persons model migrator (introduced in Sect. 2.2.1) that provides such a method. Basically, we move the migration configuration to the constructor and provide a new method called `isMigrationNeeded`, which uses the corresponding method of the `EdeltaVersionMigrator` class.

Listing 18 Alternative implementation of the persons model migrator providing also a method to check if migration is needed.

```
1  public class PersonsModelMigrator {
2
3      private EdeltaVersionMigrator edeltaMigrator;
4
5      public PersonsModelMigrator(String modelPath) {
6          edeltaMigrator = new EdeltaVersionMigrator();
7          edeltaMigrator.loadCurrentEPackage(PersonsPackage.eINSTANCE);
8          edeltaMigrator.registerMigration(FromV1ToV2::new);
9          // other registered migrations...
10         edeltaMigrator.addModelFileExtension(".persons");
11         edeltaMigrator.loadModelsFrom(modelPath);
12     }
13
14     public boolean isMigrationNeeded() {
15         return edeltaMigrator.isMigrationNeeded();
16     }
17
18     public Collection<Resource> execute() throws Exception {
19         return edeltaMigrator.execute();
20     }
21 }
```

4.4 How Edelta is developed and tested

Throughout this paper, we frequently emphasize the importance of automated testing for the migration logic and its integration into the Eclipse user interface. In that respect, the proposed approach is based on the same strategies used to develop the Edelta framework and its DSL. We have relied on automated tests since the beginning of the implementation of Edelta and its DSL. We have several unit tests for the Edelta runtime library and the compiler of the Edelta DSL. Moreover, we have integration and UI tests for all the Edelta features related to Eclipse (in particular, the integration of the Edelta DSL compiler in Eclipse). For each commit on the Edelta GitHub repository, which also includes our refactorings library, 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[19]) and static analysis using SonarCloud[20] (a cloud service based on the well-known code quality analysis tool SonarQube).

5 Related work

Co-evolution approaches in MDE can be divided into three types: inference approaches, manual approaches, and operator approaches [9].

An *inference approach* compares or differentiates the original and updated metamodel to automatically generate evolutionary strategies for models. Cicchetti et al. take a model containing the metamodel evolution as input and generate a model transformation through a higher-order model transformation for migrating the models [27].

A *manual* approach allows designing strategies, usually by hand, for migrating models to an updated metamodel. Taentzer et al. defined a manual approach that employs algebraic graph transformation rules to define model migrations [28]. The rules derive from the metamodel evolution rules and can be customized to fit the requirements. Paternostro et al. proposed Ecore2Ecore, which automatically generates a partial migration strategy from defined interrelationships between original and evolved metamodels [29]. However, the approach only supports simple evolution patterns.

[19] https://coveralls.io/

[20] https://www.sonarsource.com/products/sonarcloud/

An *operator approach* leverages a specific language that allows defining model migration strategies. Our framework is an example of an operator-based approach. Another example is Epsilon Flock, which uses a specific transformation language for model migration [30, 31]. Users define migration rules for evolved model elements. Flock takes care of the rest, using a conservative copy algorithm to duplicate unchanged model elements and apply the defined rules automatically. Also, our framework automatically duplicates these unchanged elements, ensuring the migration strategy's definitions remain succinct. Moreover, Edelta automatically executes straightforward metamodel migrations, such as deletions and renamings. Another interesting approach similar to ours is COPE [32, 33]. COPE records metamodel evolution changes in a "history" file. Users then build a migrator from this history, employing a set of primitives (like create, delete, or rename elements) and reusable migration patterns for metamodel adaptation and model migration. COPE also allows custom migration strategies through the Groovy language. Our framework offers a simpler, less error-prone alternative: the Edelta DSL is statically typed and uses type inference for declarations. Edelta generates the migrator immediately upon compilation, unlike COPE, which requires several preliminary steps.

The literature offers various model migration approaches, including search-based [34] and feature-based [35] methods, and some also focus on concrete syntax and constraints. For instance, Kessentini et al. propose an interactive multi-objective algorithm that dynamically adapts models as metamodels evolve [36]. The approach also gives developers interactive suggestions for editing operations to apply their operators. Another notable co-evolution approach, built on the MetaEdit+ platform, illustrates a holistic management strategy that automatically updates models or prompts human intervention when necessary [37].

Some characteristics that should be considered when developing approaches for supporting model conformance when metamodels evolve are the *correctness* of the resulting models with *automation*, and the *tool support* to assist developers in selecting between the identified methods. Hebig et al. surveyed 31 approaches to provide insights and trade-offs in choosing the right approaches [10].

The correctness can be syntactic or semantic. In the first case, if models serve as documentation, simply ensuring their accessibility might be enough. It means ensuring they are syntactically conformant to the new metamodel so updated editors can open them. For example, if the lower bound of a multiplicity increases, it could create references and objects like the approach by Taentzer et al. [28]. Regarding syntactical metamodel conformance, our framework and all 31 approaches can be used. Conversely, when models serve as input (like for code generation), simply retrieving syntactically conforming models is not enough. The goal is to retrieve a semantically "correct" model. For instance, if a type restriction is added to a metamodel property, standard solutions such as those of Meyers et al. [38] and COPE/Edapt[21] [33] that would eliminate all instances of non-compliant properties are not sufficient. This emphasizes a crucial point. When a generic solution is insufficient, approaches should provide users some control over the resolution's outcome to achieve desired results. Many approaches studied in the survey support the users during model migrations. The survey classifies the approaches based on whether generalization resolution is possible or not per metamodel change within the given domain, company, or project. In the first case, the following approaches can be used: COPE/Edapt [33], Krause et al. [39], Didonet et al. [40], and Wittern [41], as well as by combining change identification, like Williams et al. [34], Vermolen et al. [42], and Muller et al. [43], with the resolution approaches, like Wimmer et al. [44], EMFMigrate [45], Epsilon Flock [31], and Meyers et al. [38]. If generalization is not possible, approaches such as Cross-Layer Modeler [46], CARE [47], Kessentini et al. [48], and GraCoT [49] can be exploited. Our approach allows migration strategies to be defined using an ad hoc DSL so that users can explicitly define how to migrate models.

The second significant trade-off when choosing an approach is automation support, primarily because manual effort drives co-evolution's running costs. While making decisions per model offers high control over the outcome, this can become unfeasible when hundreds of models need co-evolution. Many approaches, such as Garcés et al. [50], COPE/Edapt [33], Williams et al. [34], and others, can be adopted if the main goal is to automate the resolution fully. However, if the aim is to reduce manual effort, many approaches can be applied like [50], COPE/Edapt [33], EMFMigrate [45], Cross-Layer Modeler [46], CARE [47], and Kessentini et al. [48]. Our approach can be used in both cases, i.e., to maximize the automatic resolution and minimize the manual effort.

The selected approaches in the Hebig et al. survey are often partially implemented, and some implementations are publicly available [10]. This is a key factor for companies with low investment ability, as reusing an existing tool makes an approach significantly easier to adopt. Unlike the presented approaches, our framework provides both the publicly available implementation and a development process to provide a reliable way to migrate models in different scenarios.

6 Conclusion

Metamodel co-evolution is a tedious activity needed to keep metamodels, artifacts, and tools consistent in response to evo-

[21] https://projects.eclipse.org/projects/modeling.edapt

lution. In real-world scenarios, developers have little control over client-created models, and keeping consistent model-based applications in case of evolution requires additional effort.

In this paper, we significantly advanced the capabilities of Edelta to handle the migration of client models also in the presence of mutually dependent metamodels and models, which may be an essential requirement in complex modeling ecosystems. To demonstrate the practicality of the enhancements, we introduced new sophisticated examples and also explored various strategies for integrating model migration into client-side Eclipse applications, offering concrete guidance for the Edelta adoption. To show a broader applicability example, we showed how the migration logic can operate outside the Eclipse environment, exemplified by a standalone REST service implementation. This enables a wider range of deployment contexts and integration with existing model repositories' approaches. Furthermore, we proposed an operation workflow that guides developers through the entire development process, from specifying co-evolution scenarios and writing migration code, to deploying and executing migrations in both Eclipse-based and REST-based client applications. The workflow also incorporates automated testing practices, including unit tests for migration code and functional UI integration tests.

We discussed the development and application of our approach by mentioning its strengths and areas for improvement. This comprehensive evaluation underscores the practical value and versatility of Edelta in supporting complex model evolution scenarios in diverse contexts. Future works and directions not only include other possible intricate and complex migration strategies [15, 22], but also improving and generalizing the REST service scenario. Indeed, there is room for further automation in order to support live metamodel evolution without the need to redeploy the REST service to align with new metamodels and possible evolutions.

Our approach, based on the Edelta framework, is inherently tied to the Eclipse Modeling Framework (EMF) ecosystem. The general concepts and development workflows we presented might be adapted to other modeling frameworks. However, the whole Edelta concepts (runtime libraries and DSL) should first be re-implemented for the target modeling framework, which will require significant effort depending on the framework's capabilities and architecture.

A Another refactoring for the Library example

We present a third refactoring applied to the second example. We continue from where we left off in Sect. 3.3.1, after the second migration of the library models.

Listing 19 The Edelta program for the third migration of the library example.

```
import edelta.refactorings.lib.EdeltaRefactorings

metamodel "ecore"

migrate "http://edelta/Library/v3" to "http://edelta/Library
    /v4"
migrate "http://edelta/Books/v3" to "http://edelta/Books/v4"

use EdeltaRefactorings as refactorings

modifyEcore compactItems epackage library {
  ecoreref(BookItem).addNewEAttribute("numOfCopies", ecoreref(EInt)) [
    defaultValue = 1
  ]
}
```

In the following, we will use as input models in our tests the four models shown in Fig. 21. Note that we intentionally have the first library model with book items referring to the same book.

However, we would like to store the information about the number of copies of a book in a library, instead of having several `BookItem` objects referring to the same `Book` object. Moreover, currently, we have no way to specify that we have 0 copies of a book in a library (i.e., all available copies have been borrowed).

For this reason, we create a new refactoring, shown in Listing 19, which adds a new attribute `numOfCopies` to the `BookItem` class, which will be used to store the number of copies of a book in a library.

We import the metamodel `ecore` to use the `EInt` type. We also set the default value of the new attribute `numOfCopies` to 1, so that when we create a new `BookItem` object, it will have 1 copy of the book by default. That is because we assume that when we create a new `BookItem` object, we have just bought at least one copy of the book in the library. Of course, that is just a design decision, and we could have set the default value to 0.

We update the migrator to register this new migration. We also create a new unit test using the four models shown in Fig. 21 and verify that the output is as expected.

Now, the users can set the new attribute `numOfCopies` in the properties view of the `BookItem` element, instead of duplicating the `BookItem` objects. However, existing `Library` models must be updated manually: the number of duplicated `BookItem` objects must be merged into a single `BookItem` object with the correct value of `numOfCopies`, and the other duplicate copies must be manually removed. This is, of course, cumbersome for the users and error-prone, especially for large models. To address this, we further automate the migration process using Edelta, ensuring that all existing models are updated consistently and efficiently.

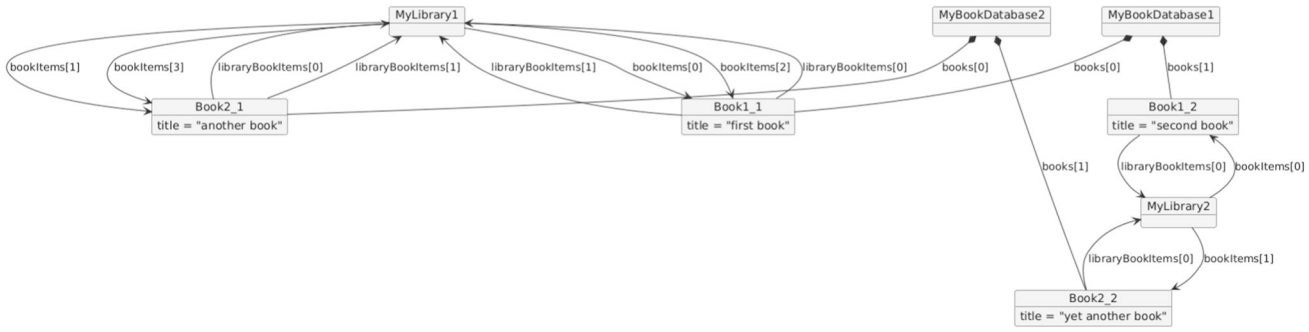


Fig. 21 The four models after the second migration

Listing 20 The Java implementation of the validation constraint for duplicated `BookItem` objects.

```
1  public class LibraryValidator extends EObjectValidator {
2    public boolean validateLibrary_noDuplicateBooks(Library library,
         DiagnosticChain diagnostics, Map<Object, Object> context) {
3      boolean valid = true;
4
5      Map<Book, List<BookItem>> bookToItems = library.getBookItems().stream()
6        .collect(Collectors.groupingBy(item -> item.getBook()));
7
8      for (Map.Entry<Book, List<BookItem>> entry : bookToItems.entrySet()) {
9        List<BookItem> items = entry.getValue();
10       if (items.size() > 1) {
11         valid = false;
12         diagnostics.add(new BasicDiagnostic(
13           Diagnostic.ERROR,
14           DIAGNOSTIC_SOURCE,
15           0,
16           "Duplicate BookItem refer to Book '"+entry.getKey().
       getTitle()+"'",
17           items.toArray()));
18       }
19     }
20
21     return valid;
22   }
23 }
```

Moreover, we should consider duplicated `BookItem` objects as errors from now on. In particular, we add a new validation constraint to check that there are no duplicated `BookItem` objects in a `Library` model. Such a constraint cannot be specified in the Ecore directly. We can use one of the mechanisms provided by EMF to specify such a constraint. We refer the interested reader to the EMF documentation for more information on how to specify validation constraints. In Listing 20, we only show the implementation of the constraint in Java, which is used by the EMF validation mechanisms.

The test for the input models with duplicated `BookItem` objects now fails, as expected:

```
java.lang.AssertionError: Diagnosis of library.impl.LibraryImpl@
...edelta.example.migration.library.tests/output/MyLibrary1.library#/
Duplicate BookItems refer to Book 'another book'
Duplicate BookItems refer to Book 'first book'
```

By enabling live validation in the EMF editor, we can see the errors in the editor as well, as shown in Fig. 22.

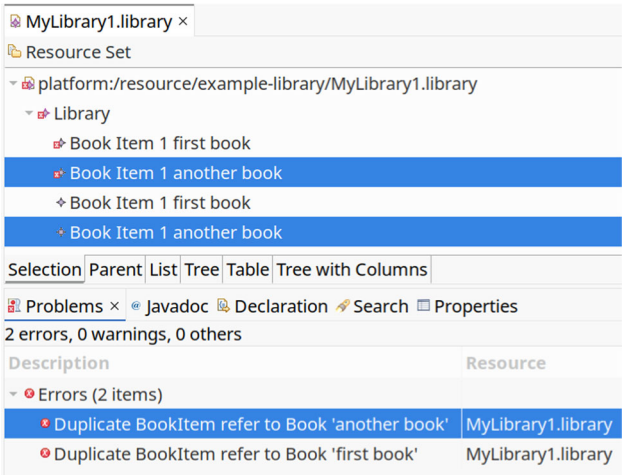


Fig. 22 The four models after the second migration

Let's fix the problem by modifying the previous Edelta program as shown in Listing 21, implementing a model migration that merges the duplicated `BookItem` objects into a single one with the correct value of `numOfCopies` and removes the other duplicate copies. We add a model migration rule after the previous migration, so that it is applied after the new attribute `numOfCopies` has been added to the `BookItem` class. We declare variables before the `modelMigration` block to refer to the features we are interested in. The lambda expression is closed with respect to such variables, which are ensured to refer to the new metamodel(s) during the model migration.

The idea is to perform operations similar to those of the validator. However, this time, we must use the reflective API, and we have to deal with the original objects, which are not the same as the ones in the migrated model. Thus, the implementation is a bit more complex than that of the validator, not to mention that now we must also take care of the migration of the objects. We must be careful to use the `getOriginal` and `getMigrated` methods to refer to the original and migrated objects, respectively. In particular, `getMigrated` also takes care of migrating the corresponding object, in case it has not been migrated yet.

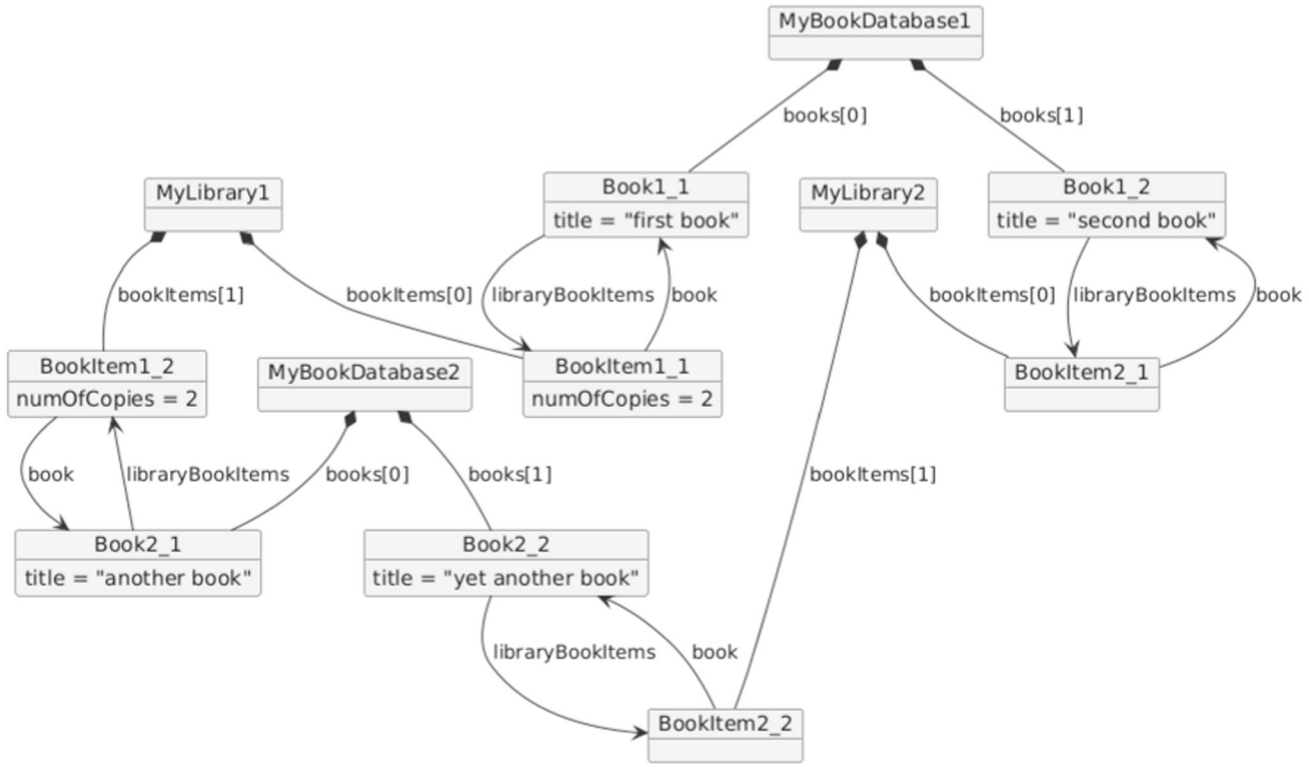


Fig. 23 The four models after the second migration with proper compacting

Listing 21 The Edelta program for the third migration of the library example with model migration.

```
modifyEcore compactItems epackage library {
  ecoreref(BookItem).addNewEAttribute("numOfCopies", ecoreref(EInt)) [
    defaultValue = 1
  ]

  val bookItemsFeature = ecoreref(library.Library.bookItems)
  val bookFeature = ecoreref(library.BookItem.book)
  val numOfCopiesFeature = ecoreref(BookItem.numOfCopies)
  modelMigration[
    copyRule(
      wasRelatedTo(bookItemsFeature),
      [oldBookItemsFeature, oldLibrary, newLibrary |
        val oldBookItems = oldLibrary.getValueAsList(oldBookItemsFeature)

        val groupedByBook = oldBookItems.stream
          .collect(Collectors.groupingBy(
            // group by the old bookFeature
            [oldBookItem|
              oldBookItem.getValueAsEObject(getOriginal(bookFeature))]))

        val newBookItems = groupedByBook.entrySet.stream
          .map[entry|
            // create a single copy for the old items referring to the same book
            // (take the first one since they are all the same)
            var oldBookItem = entry.value.head;
            createFrom(getMigrated(oldBookItem.eClass), oldBookItem) [
              // set the new counting value to the number of items in the old model
              eSet(numOfCopiesFeature, entry.value.size)
            ]
          ].toList
        // set the new book items in the new library
        newLibrary.eSet(bookItemsFeature, newBookItems)
      ]
    )
  ]
}
```

Listing 22 The Edelta program for the third migration of the library example with model migration using a reusable migration rule.

```
modifyEcore compactItems epackage library {
  ecoreref(BookItem).addNewEAttribute("numOfCopies", ecoreref(EInt)) [
    defaultValue = 1
  ]

  val bookItemsFeature = ecoreref(library.Library.bookItems)
  val bookFeature = ecoreref(library.BookItem.book)
  val numOfCopiesFeature = ecoreref(BookItem.numOfCopies)
  modelMigration[
    copyGroupingCountingRule(
      bookItemsFeature,
      bookFeature,
      numOfCopiesFeature
    )
  ]
}
```

However, our refactoring library provides several reusable migration rules for recurrent situations. In particular, we provide one for the case above, which groups and counts the duplicated `BookItem` objects and sets the correct value for `numOfCopies`.

The Edelta program can be simplified as shown in Listing 22 by using the reusable migration rule `copyGroupingCountingRule` from our refactoring library (with the same semantics). With this new implementation, we can run the Edelta program and verify that the output models are as expected. In particular, the duplicated `BookItem` objects

are merged into a single one with the correct value of `numOfCopies`, and the validator does not detect any problem. This demonstrates how Edelta can automate not only metamodel evolution but also complex model migrations that would otherwise require significant manual effort.

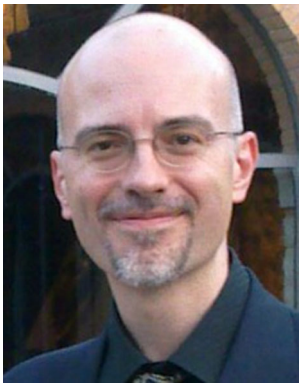
The resulting models are shown in Fig. 23, confirming the correctness and effectiveness of the migration.

Acknowledgements This work was partially supported by the MUR (Italy) Department of Excellence 2023 - 2027 for GSSI, by the European HORIZON-KDT-JU research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems", HORIZON-KDT-JU-2023-2-RIA, Proposal number: 101140216-2, KDT232RIA_00017, by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY – CUP: D13C21000430001, and by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem, grant PE0000020 – CHANGES–CUP: D53C22002560006.

References

1. Iovino, L., Pierantonio, A., Malavolta, I.: On the impact significance of metamodel evolution in MDE. J. Object Technol. **11**(3) (2012)
2. Pérez, J.G.: Maintainability of transformations in evolving mde ecosystems. PhD thesis, Universidad del País Vasco-Euskal Herriko Unibertsitatea (2014)
3. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Model Comparison in Practice, pp. 30–38. ACM (2011)
4. Durisic, D., Staron, M., Tichy, M., Hansson, J.: Assessing the impact of meta-model evolution: a measure and its automotive application. Softw. Syst. Model. **18**(2), 1419–1445 (2019)
5. Sousa Saraiva, J., Silva, A.R.: Evaluation of MDE tools from a metamodeling perspective. J. Database Manag. (JDM) **19**(4), 21–46 (2008)
6. Di Ruscio, D., Lämmel, R., Pierantonio, A.: Automated Co-evolution of GMF Editor Models. In: Software Language Engineering, pp. 143–162. Springer, Berlin, Heidelberg (2011)
7. Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., Schönböck, J.: Consistent co-evolution of models and transformations. In: MODELS, pp. 116–125. IEEE, Ottawa, Canada (2015)
8. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Dealing with the coupled evolution of metamodels and model-to-text transformations. In: ME@MODELS, pp. 22–31 (2014)
9. Paige, R.F., Matragkas, N., Rose, L.M.: Evolving models in model-driven engineering: state-of-the-art and future challenges. J. Syst. Softw. **111** (2016)
10. Hebig, R., Khelladi, D.E., Bendraou, R.: Approaches to co-evolution of metamodels and models: a survey. IEEE Trans. Software Eng. **43**(5), 396–414 (2016)
11. He, X., Avgeriou, P., Liang, P., Li, Z.: Technical debt in MDE: a case study on GMF/EMF-based projects. Presented at the (2016)
12. Casalaro, G.L., Cattivera, G., Ciccozzi, F., Malavolta, I., Wortmann, A., Pelliccione, P.: Model-driven engineering for mobile robotic systems: a systematic mapping study. Softw. Syst. Model. **21**(1), 19–49 (2022)
13. Basciani, F., D'Emidio, M., Di Ruscio, D., Frigioni, D., Iovino, L., Pierantonio, A.: Automated selection of optimal model transformation chains via shortest-path algorithms. IEEE Trans. Software Eng. **46**(3), 251–279 (2018)
14. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta 2.0: supporting live metamodel evolutions. In: MODELS, pp. 1–10. ACM/IEEE, New York, NY, USA (2020)
15. Bettini, L., Ruscio, D.D., Iovino, L., Pierantonio, A.: An executable metamodel refactoring catalog. Softw. Syst. Model. **21**, 1689–1709 (2022)
16. Bettini, L., Di Salle, A., Iovino, L., Pierantonio, A.: Supporting reusable model migration with Edelta. J. Syst. Softw. **212**, 112012 (2024)
17. Bettini, L., Di Ruscio, D., Di Salle, A., Iovino, L., Pierantonio, A.: Keeping clients' models up-to-date with Edelta. In: Proc. of MODELS, pp. 46–50. ACM, New York, NY, USA (2024)
18. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham, UK (2016)
19. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta: An approach for defining and applying reusable metamodel refactorings. Presented at the (2017)
20. Basciani, F., Di Ruscio, D., Iovino, L., Pierantonio, A.: Automated quality assessment of interrelated modeling artifacts. In: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 234–243 (2021). IEEE
21. Bettini, L., Ruscio, D.D., Iovino, L., Pierantonio, A.: Supporting safe metamodel evolution with Edelta. Int. J. Softw. Tools Technol. Transfer **24**, 247–260 (2022)
22. Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: SLE. LNCS, vol. 6563, pp. 163–182 (2010)
23. Basso, F.P., Oliveira, T.C., Werner, C.M., Becker, L.B.: Building the foundations for 'MDE as Service'. IET Software **11**(4), 195–206 (2017)
24. Giraudet, T., Bats, M., Blouin, A., Combemale, B., David, P.-C.: Sirius web: insights in language workbenches-an experience report. J. Object Technol. (2024)
25. Basciani, F., Di Rocco, J., Di Ruscio, D., Di Salle, A., Iovino, L., Pierantonio, A.: MDEForge: an extensible web-based modeling platform. In: CloudMDE@ MoDELS, pp. 66–75 (2014)
26. López, J.A.H., Canovas Izquierdo, J.L., Cuadrado, J.S.: ModelSet: a dataset for machine learning in model-driven engineering. Softw. Syst. Model. **21**(3), 967–986 (2022)
27. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: In: In: EDOC, (ed.) Automating co-evolution in model-driven engineering, pp. 222–231. IEEE, USA (2008)
28. Taentzer, G., Mantz, F., Arendt, T., Lamo, Y.: Customizable model migration schemes for meta-model evolutions with multiplicity changes. In: MODELS, pp. 254–270. Springer (2013)
29. Paternostro, M., Hussey, K.: Advanced features of the eclipse modeling framework. EclipseCON (2006)
30. Rose, L., Kolovos, D., Paige, R., Polack, F.: Model migration with epsilon flock. In: Theory and Practice of Model Transformations, pp. 184–198. Springer (2010)
31. Rose, L., Kolovos, D., Paige, R., Polack, F., Poulding, S.: Epsilon flock: a model migration language. Softw. Syst. Model. **13**(2), 735–755 (2014)
32. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: COPE - automating coupled evolution of metamodels and models. In: ECOOP. LNCS, vol. 5653, pp. 52–76. Springer, Berlin, Heidelberg (2009)
33. Herrmannsdoerfer, M.: In: SLE. LNCS, (ed.) COPE - a workbench for the coupled evolution of metamodels and models, vol. 6563, pp. 286–295. Springer, Berlin, Heidelberg (2010)
34. Williams, J., Paige, R., Polack, F.: Searching for model migration strategies. In: Workshop on Models and Evolution. ME12, pp. 39–44. ACM, New York, NY, USA (2012)
35. Di Ruscio, D., Etzlstorfer, J., Iovino, L., Pierantonio, A., Schwinger, W.: Supporting variability exploration and resolution

during model migration. In: ECMFA@STAF. LNCS, vol. 9764, pp. 231–246. Springer, Berlin, Heidelberg (2016)
36. Kessentini, W., Wimmer, M., Sahraoui, H.: Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search. In: MODELS, pp. 101–111 (2018)
37. Kelly, S., Tolvanen, J.: How MetaEdit+ supports co-evolution of modeling languages, tools and models. In: MODELS, pp. 9–13. IEEE, New York, NY, USA (2023)
38. Meyers, B., Wimmer, M., Cicchetti, A., Sprinkle, J.: A generic in-place transformation-based approach to structured model co-evolution. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **42** (2011)
39. Krause, C., Dyck, J., Giese, H.: Metamodel-specific coupled evolution based on dynamically typed graph transformations. In: ICMT@STAF 2013. LNCS, vol. 7909, pp. 76–91. Springer, Berlin, Heidelberg (2013)
40. Fabro, M.D.D., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: SAC, pp. 963–970. , ACM, New York, NY, USA (2007)
41. Wittern, H.: In: EDOC Workshops, (ed.) Determining the Necessity of Human Intervention When Migrating Models of an Evolved DSL, pp. 209–218. IEEE, Los Alamitos, CA, USA (2013)
42. Vermolen, S.D., Wachsmuth, G., Visser, E.: Reconstructing complex metamodel evolution. In: Software Language Engineering, pp. 201–221. Springer, Berlin, Heidelberg (2012)
43. Müller, K., Rumpe, B.: User-driven adaptation of model differencing results. Softwaretechnik-Trends **34**(2) (2014)
44. Wimmer, M., Kusel, A., Schönböck, J., Retschitzegger, W., Schwinger, W., Kappel, G.: On using inplace transformations for model co-evolution. In: Proc. 2nd Int. Workshop Model Transformation with ATL, vol. 711, pp. 65–78 (2010). Citeseer
45. Wagelaar, D., Iovino, L., Di Ruscio, D., Pierantonio, A.: In: Translational semantics of a co-evolution specific language with the emf transformation virtual machine, pp. 192–207. Springer (2012)
46. Demuth, A., Riedl-Ehrenleitner, M., Lopez-Herrejon, R.E., Egyed, A.: Co-evolution of metamodels and models through consistent change propagation. J. Syst. Softw. **111**, 281–297 (2016)
47. Schoenboeck, J., Kusel, A., Etzlstorfer, J., Kapsammer, E., Schwinger, W., Wimmer, M., Wischenbart, M.: Care: a constraint-based approach for re-establishing conformance-relationships. In: Proceedings of the Tenth Asia-Pacific Conference on Conceptual Modelling-Volume 154, pp. 19–28 (2014)
48. Kessentini, W., Sahraoui, H., Wimmer, M.: In: Automated metamodel/model co-evolution using a multi-objective optimization approach, pp. 138–155. Springer (2016)
49. Gómez, P., Sánchez, M.E., Florez, H., Villalobos, J.: An approach to the co-creation of models and metamodels in enterprise architecture projects. J. Object Technol. **13**(3), 2–129 (2014)
50. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: In: Managing model adaptation by precise detection of metamodel changes, pp. 34–49. Springer (2009)



Lorenzo Bettini 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 with IDE support (in particular Object-Oriented languages and Network aware languages). He is a committer of the Eclipse projects Xtext and SWTBot, and the project lead of the Eclipse project EMF Parsley. Contact him at lorenzo.bettini@unifi.it, or visit https://www.lorenzobettini.it.

Amleto Di Salle Amleto Di Salle is currently an assistant professor (tenure track) at the Gran Sasso Science Institute (Italy) and previously a research fellow at the European University of Rome (Italy). In 2015, he received a Ph.D. in computer science from the University of L'Aquila. His main research focuses on several aspects of software engineering, particularly in software architecture, model-based software engineering, distributed systems composition, and software systems maintenance and evolution, with a focus on technical debt and code smells. He published more than 60 papers in international conferences and journals. He has worked on several European and national research projects, such as MATISSE, CHOReVOLUTION, CHOReOS, INCIPICT, and PRIN 2022 COBOL. He is currently an associate editor of the Journal of Computer Languages.

Ludovico Iovino is currently an Associate Professor at the Gran Sasso Science Institute, Computer Science scientific Area. His interests include Software Engineering, Model Driven and Low-code Development. He was PI for various scientific and technology transfer projects with topics ranging from the application of IoT, Web Engineering, Infomobility, and Modeling for Digital Twin. He's also part of the executive board of the national consortium for computer sciences and leads the Smart City Laboratory at his institute. Contact him at ludovico.iovino@gssi.it or visit https://www.ludovicoiovino.it/