COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Zohra Kaouter KEBAILI

Automated co-evolution of metamodels and code

« Sous-titre de la thèse »

Thèse présentée et soutenue à « Lieu », le « date »
Unité de recherche : IRISA -UMR 6074-
Thèse N° : « si pertinent »

Rapporteurs avant soutenance :

Prénom NOM Fonction et établissement d'exercice
Prénom NOM Fonction et établissement d'exercice
Prénom NOM Fonction et établissement d'exercice

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	Prénom NOM	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
	Prénom NOM	Fonction et établissement d'exercice
Dir. de thèse :	Prénom NOM	Fonction et établissement d'exercice
Co-dir. de thèse :	Prénom NOM	Fonction et établissement d'exercice *(si pertinent)*

Invité(s) :

Prénom NOM Fonction et établissement d'exercice

Acknowledgement

Je tiens à remercier
I would like to thank. my parents..
J'adresse également toute ma reconnaissance à ....
....

Table of Contents

List of Figures

List of Tables

Résumé en Français

Contexte: Les systèmes logiciels deviennent de plus en plus complexes, ce qui engendre une charge importante en matière de maintenance, souvent accompagnée de coûts élevés pouvant même dépasser ceux liés au développement initial [1]. En 2001, le groupe Object Management Group (OMG) a introduit une approche l'Ingénierie Dirigée par les Modèles (IDM) [2]. Depuis, cette méthode s'est largement répandue dans le développement et la maintenance des systèmes de grande envergure ainsi que des logiciels embarqués. Elle permet également d'optimiser la productivité des concepteurs. Son adoption permet également aux entreprises de réduire les délais (tant pour la conception que pour la mise sur le marché), de diminuer les dépenses (liées au développement, à l'intégration et à l'adaptation) et d'améliorer la durabilité ainsi que la compétitivité à l'échelle internationale. l'Ingénierie Dirigée par les Modèles (IDM) utilise les modèles comme entité d'abstraction de haut niveau. Ces modèles sont utilisés comme une entrée pour la génération de plusieurs artéfacts : contraintes, transformations, instances de modèles, et le code. Dans le cadre de cette thèse, nous nous intéressons particulièrement au code. Le code généré à partir des modèles est ensuite enrichi pour ajouter d'autres fonctionnalités. Par exemple, le projet JHipster propose de générer, à partir de modèles d'entités, les différentes parties d'applications web modernes du côté client et serv eur. Un autre exemple populaire est celui d'OpenAPI, où de nombreux artefacts sont générés à partir d'une spécification d'API. Certaines plates-formes de développement low-code s'appuient sur certains des principes de l'IDM, avec des modèles de données et d'abstraction. Dans la plateforme d'Eclipse, le Framework de Modélisation Eclipse (EMF) est un autre exemple important. Sur la base d'un métamodèle, EMF génère des APIs de code Java, des adaptateurs et un éditeur. Ce code généré est ensuite enrichi par les développeurs pour offrir des fonctionnalités et des outils supplémentaires, tels que la validation, la simulation ou le débogage, etc. Par exemple, les implémentations Eclipse de UML [1] et BPMN [2] s'appuient sur les métamodèles UML et BPMN pour générer leur API de code correspondante. Par la suite, l'ensemble des outils et services complémentaires est développé à partir de cette API.

1. `https://www.eclipse.org/modeling/mdt/downloads/?project=uml2`
2. `https://www.eclipse.org/bpmn2-modeler/`

Problématique: 1) L'évolution des modèles impacte les artéfacts générés ce qui révèle un défi important sur leur co-évolution. Parmi ces artefacts, le code est particulièrement concerné. En effet, lorsqu'un métamodèle change et que l'API de code est régénérée, le code supplémentaire développé par les programmeurs peut être affecté. Il est alors essentiel d'assurer son adaptation pour maintenir la cohérence avec les nouvelles versions du métamodèle. Toutefois, si la co-évolution est réalisée manuellement, elle reste coûteuse en temps et en effort et est susceptible de provoquer des erreurs. Afin de surmonter ces difficultés, les chercheurs ont exploré, depuis plusieurs décennies, différentes approches visant à automatiser ce processus. Ces travaux ne se sont pas limités au code, mais ont concerné l'ensemble des artefacts de l'écosystème MDE, en mettant l'accent sur l'automatisation. D'autres aspects, comme l'optimisation et la résilience à l'évolution du métamodèle, ont également été étudiés. De nombreuses recherches ont abordé la question de la co-évolution en MDE. Certains travaux [4]–[9] se sont concentrés sur la vérification de la cohérence entre les modèles et le code, sans pour autant traiter leur co-évolution. D'autres études [10], [11] ont proposé des solutions permettant d'adapter le code aux évolutions du métamodèle, mais avec certaines limites. Certaines approches se focalisent uniquement sur l'API de code générée, sans prendre en compte le code additionnel, et se contentent de maintenir une traçabilité bidirectionnelle entre le modèle et l'API. D'autres offrent une co-évolution semi-automatique nécessitant l'intervention des développeurs, mais sans mécanisme de validation après la co-évolution, ni de comparaison avec une référence [11], [127], [129]–[131], [133], [147], [155], [203].

Dans cette thèse, nous proposons une approche automatisée pour gérer l'impact de l'évolution des métamodèles sur le code, accompagnée d'un processus de validation permettant de valider la co-évolution.

2) Dans tout système reposant sur l'Ingénierie Dirigée par les Modèles, les éléments du métamodèle sont utilisés dans le code API et puis dans le code additionnel. L'évolution du métamodèle sera propagée dans le code co-évolué et son comportement peut être altéré. D'où l'importance de vérifier la l'exactitude de la co-évolution.

Dans une perspective plus large, peu de travaux ont été consacrés à la vérification de la correction de l'évolution du code en général. Ge et al. [12] proposent une méthode pour vérifier l'xexactitude des refactorisations. Hors du cadre de l'évolution du code, on trouve des approches de sélection de tests incrémentiels telles que Infinitest [3], EKSTAZI [13] et Moose [14]. Ces outils, Infinitest, EKSTAZI et Moose, ont pour objectif d'analyser les

3. `https://infinitest.github.io/doc/eclipse`

modifications du code de manière incrémentale afin de sélectionner uniquement les tests impactés dans la version évoluée du code.

3) Dans l'écosystème de l'Ingénierie Dirigée par les Modèles (IDM), la co-évolution des métamodèles et du code peut être considérée comme l'une des nombreuses tâches existantes, aux côtés de la génération de modèles et de la génération de code. Depuis leur apparition, les modèles de langage de grande taille (abrégé LLM de l'anglais large language model) ont été appliqués à divers domaines de la recherche scientifique, notamment en ingénierie logicielle et en IDM. Cependant, à notre connaissance, l'exploration des LLMs pour la co-évolution des métamodèles et du code n'a pas encore été abordée.

Dans le domaine de l'Ingénierie Logicielle, Fu et al. [15] ont évalué la capacité de ChatGPT à détecter, classer et corriger du code vulnérable. Kabir et al. [16] ont étudié son aptitude à générer du code et à le maintenir en l'améliorant sur la base de nouvelles spécifications fonctionnelles. Zhang et al. [17] ont proposé CodEditor, un outil basé sur les LLMs pour la co-évolution du code entre différents langages de programmation. Cet outil apprend les évolutions sous forme de séquences d'édition et utilise des LLMs pour assurer une traduction multilingue du code.

Par ailleurs, d'autres travaux ont évalué l'usage des LLMs dans les activités de IDM. Chen et al. [18] et Camara et al. [19] ont exploré l'utilisation de ChatGPT pour la génération de modèles. Chaaben et al. [20] ont démontré que l'apprentissage par quelques exemples (few-shot learning) avec le modèle GPT-3 pouvait être efficace pour la complétion de modèles et d'autres activités de modélisation.

Contributions: Pour répondre à ces problème, je propose les contributions suivantes: 1) Tout d'abord, je propose une approche entièrement automatisée de co-évolution du code en réponse à l'évolution des métamodèles. Cette approche est guidée par le compilateur. Elle est basée sur la correction des erreurs de compilation causées par l'évolution du métamodèle. Les erreurs sont analysées afin de les associer à des patrons d'utilisation des éléments du métamodèle qui ont évolué. Suite à cela, on arrive à sélectionner une ou plusieurs résolutions à appliquer pour chaque erreur (dépendamment du nombre du changements du métamodèle impactants), et donc co-évoluer les erreurs du code automatiquement.

L'évaluation de cette approche a été réalisée sur neuf projets Eclipse issus d'OCL, Modisco et Papyrus, selon quatre axes : 1) Évaluation de la correction de la co-évolution à l'aide de tests unitaires générés automatiquement. 2) Vérification de la correction comportementale en exécutant des tests unitaires avant et après la co-évolution automatique

du code. 3) Comparaison avec l'approche de co-évolution semi-automatique de référence [11]. 4) Comparaison avec l'outil populaire de corrections rapides (Quick Fixes). Un prototype sous forme de plugin Eclipse a été mis en ligne (lien). Les résultats obtenus montrent que notre approche atteint une précision moyenne de 82% et un rappel moyen de 81%, avec des variations allant de 48% à 100% pour la précision et le rappel. 2) Ma seconde contribution est une approche visant à aider les développeurs à vérifier l'exactitude de la co-evolution, autrement dit, que la co-évolution n'a pas altéré le comportement du code. Mon approche exploite les tests unitaires avant et après la co-évolution et génère un rapport visuel indiquant les tests réussis, échoués et erronés avant et après l'évolution du code. Ce rapport permet d'obtenir une meilleure compréhension de la co-évolution et de son impact sur le code. Avant de procéder à une évaluation quantitative, une étude utilisateur a été menée afin d'évaluer la difficulté de tracer manuellement des tests impactés après l'évolution du métamodèle et la co-évolution du code, de plus la difficulté d'investiguer l'éxactitude de la co-évolution. J'ai évalué notre approche sur 18 projets Eclipse issus d'OCL, Modisco, Papyrus et EMF, en utilisant à la fois des tests unitaires générés automatiquement et des tests écrits manuellement pour deux projets OCL. L'analyse quantitative de son utilité a révélé une réduction de 88% du nombre de tests exécutés et une diminution de 84% du temps d'exécution. Un prototype sous forme de plugin Eclipse est disponible en ligne (lien).

3) Dans ma dernière contribtion, j'explore la capacité des modèles de langage de grande taille (LLMs) à proposer des co-évolutions correctes dans le cadre de l'évolution conjointe des métamodèles et du code. Un prototype sous forme de plugin Eclipse est disponible en ligne (lien).

Notre approche a été évaluée à l'aide de ChatGPT version 3.5 sur sept projets Eclipse issus des métamodèles évolués d'OCL et Modisco. L'évaluation a pris en compte plusieurs paramètres : la variation de la température, la structure des invites (prompts) et une comparaison avec les corrections rapides (Quick Fixes) des IDE, utilisées comme référence.

Les résultats montrent que ChatGPT parvient à co-évoluer correctement 88,7% des erreurs, avec un taux de correction variant entre 75% et 100%. En modifiant les invites, nous avons observé une amélioration de la correction dans deux variantes et une diminution dans une autre. De plus, la variation du paramètre de température a révélé de meilleurs résultats avec des valeurs plus faibles. Enfin, nous avons constaté que les co-évolutions générées par ChatGPT surpassent largement les corrections rapides des IDE.

Chapter 1

Introduction

1.1 Context and Motivation

Software systems are increasingly growing in complexity, which leads to a substantial burden in terms of maintenance, often resulting in a high cost that may surpass the cost of software development itself [1]. Since Object Management Group (OMG) has introduced Model Driven Engineering in 2001 [2], MDE has been prominent in developing and maintaining large-scale and embedded systems while increasing the developers' productivity. By adopting MDE, industry can reduce time (development time and time-to-market), costs (development, integration, and reconfiguration), and improve sustainability and international competitiveness. In MDE, metamodel is a central artifact for building software languages [3]. It specifies the domain concepts, their properties, and the relationship between them. A metamodel is the cornerstone to generate model instances, constraints, transformations, and code when building the necessary language tooling, e.g. editor, checker, compiler, data access layers, etc. In particular, metamodels are used as inputs for complex code generators that leverage the abstract concepts defined in metamodels. The generated code API is used for creating, loading and manipulating the model instances, adapters, serialization facilities, and an editor, all from the metamodel elements. This generated code is further enriched by developers to offer additional functionalities and tooling, such as validation, transformation, simulation, or debugging. For instance, UML [1] and BPMN [2] Eclipse implementations rely on the UML and BPMN metamodels to generate their corresponding code API before building around it all their tooling and services in the additional code.

1. `https://www.eclipse.org/modeling/mdt/downloads/?project=uml2`
2. `https://www.eclipse.org/bpmn2-modeler/`

1.2 Challenges

C1: Resolve the impact of the metamodel evolution on the code automatically

One of the foremost challenges to deal with in MDE is the impact of the evolution of metamodels on its dependent artifacts. We focus on the impact of metamodels' evolution on the code. Indeed, when a metamodel evolves and the code API is regenerated again, the additional code implemented by developers can be impacted. As a consequence, this additional code must be co-evolved accordingly.

However, manual co-evolution can be tedious, error-prone, and time-consuming. Therefore, experts tried through last decades to find more sophistic solutions to tackle the problem of co-evolution when metamodels evolve. This interest covered almost all the artifacts in MDE ecosystem not only the code, and many solutions were proposed, inter alia, raising the automation degree of the co-evolution. Note that other aspects treated in co-evolution problem like optimization and evolution resilience. The co-evolution challenge has been extensively addressed in *MDE*. In particular, some approaches [4]–[9] focused on consistency checking between models and code, but not its co-evolution. Other works [10], [11] proposed to co-evolve the code. However, the former handles only the generated code API, it does not handle additional code and aims to maintain bidirectional traceability between the model and the code API. The latter supports a semi-automatic co-evolution requiring developers' intervention. Moreover, it does not use any validation process to check the correctness of the co-evolution and with no comparison to a baseline. In this thesis, we tackle the challenge of resolving the impact of the metamodel evolution on the code automatically followed by checking the correctness of the co-evolution.

C2: Behavioral correctness of the metamodel and code co-evolution

In literature, when the problem of metamodel and code co-evolution is addressed, the challenge of checking that the co-evolution impacted or not the behavioral correctness of the code is not handled. In any Model-Driven Engineered system, the elements of the metamodel are used in the code. Consequently, the evolution of the metamodel will be propagated in the code that is co-evolved and its behavior may be altered. Hence, the importance of checking the correctness of the co-evolution. In a larger scope, only few

works were dedicated to check the correctness of a code evolution in general. Ge et al. [12] propose to verify the correctness of refactoring. Out of the scope of testing code evolution, we find the incremental test selection approaches Infinitest [3], EKSTAZI [13], and Moose [14]. All of them aim to analyze code changes incrementally to select impacted tests in the evolved version of the code only.

C3: How to draw benefit from LLMs for the metamodel and code co-evolution

In the MDE ecosystem, we can consider that metamodel and code co-evolution is one of many other MDE tasks, like model generation and code generation. Since their appearance, Large Language Models (LLMs) have been applied in different domains of scientific research, such as Software Engineering and Model-Driven Engineering (MDE), however, to the best of our knowledge, the challenge of exploring LLMs in the task of metamodel and code co-evolution has not yet been addressed. In Software Engineering side, Fu et al. [15] evaluated ChatGPT ability to detect, classify, and repair vulnerable code. Kabir et al. [16] evaluated the ability of ChatGPT to generate code and to maintain it by improving it, based on a new feature description to add in the code. Zhang et al. [17] proposed Codeditor, an LLM based tool for code co-evolution between different programming languages. It learns code evolutions as edit sequences and then uses LLMs for multilingual translation. Moreover, other studies focused on evaluating LLMs in MDE activities. Chen et al. [18] and Camara et al. [19] used ChatGPT to generate models. Chaaben et al. [20] showed how using few-shot learning with GTP3 model can be effective in model completion and in other modeling activities.

1.3 Contributions

To tackle these three challenges, we propose three contributions:

➔ First, we propose a fully automatic code co-evolution approach due to metamodel evolution based on pattern matching. Our approach handles both atomic and complex changes of the metamodel. This approach is evaluated, on nine Eclipse projects from OCL, Modisco, and Papyrus, based on four actions: 1) Measuring the co-evolution correctness using automatically generated unit tests. 2) Verifying the behavioral

3. `https://infinitest.github.io/doc/eclipse`

correctness using unit tests running before and after automatic code co-evolution. 3) Comparing with the state-of-the art semi-automatic co-evolution approach [11]. 4) Comparing with Quick Fixes popular tool. Results show that our approach reached an average of 82% of precision and 81% of recall, varying from 48% to 100%, and from 51% to 100% for precision and recall respectively. The prototype implementation of this approach is available online as an Eclipse plugin (link).

- ➔ Second, we propose an approach that assists developers to check the behavioral correctness of the co-evolution. This approach leverages unit tests before and after the co-evolution and gives visual report about passing, failing, and erroneous tests before and after the co-evolution. This visual report allows to have more insights about the co-evolution ans its impact on the code. We then evaluated our approach on 18 Eclipse projects from OCL, Modisco, Papyrus, and EMF using both automatically generated and manually written tests. When we studied the usefulness of our approach quantitatively, we found 88% of reduction in the number of tests and 84% in execution time. The other part of the evaluation consisted of an user study experiment to gain evidence on the difficulty of the manual task of tracing impacted tests after metamodel evolution and co-evolution. The prototype implementation of this approach is available online as an Eclipse plugin (link).
- ➔ Third, we investigate the ability of LLMs in giving correct co-evolutions in the context of metamodel and code co-evolution. We evaluated our study approach with ChatGPT version 3.5 on seven Eclipse projects from OCL and Modisco evolved metamodels. the evaluation included temperature variation, prompt structure variation, and comparison with IDE Quick Fixes as baseline. Results show that ChatGPT can co-evolve correctly 88.7% of the errors, varying from 75% to 100% of correctness rate. When varying the prompts, we observed increased correctness in two variants and decreased correctness in another variant. We also observed that varying the temperature hyperparameter yields better results with lower temperatures. Finally, we found that the generated prompts co-evolutions completely outperform the quick fixes. The prototype implementation of this approach is available online as an Eclipse plugin (link).

1.4 Outline of the thesis

This manuscript is organised as follows:

- Chapter 2 provides a short background about our MDE context and main concepts that will be employed throughout the thesis.
- Chapter 3 focuses on a review of relevant studies carried out within Metamodel change detection, co-evolution in MDE ecosystem, API-client evolution, language evolution, and evolution in low-code platforms. This chapter ends with a discussion about limitations and research gap.
- Chapter 4 is devoted to our first contribution about the automatic code co-evolution approach du to metamodel evolution. It presents the algorithm of pattern matching process that selects the appropriate resolution for each error. It presents also its evaluation and results.
- Chapter 5 presents our second contribution about leveraging unit tests to check metamodel and code co-evolution behavioral correctness with its evaluation including the user experience that we conducted and results.
- Chapter 6 details the empirical study that we conducted as last contribution about exploring LLMs in metamodel and code co-evolution context, with its evaluation and results.
- Chapter 7 summarises the contributions of this work and discusses its limitations and potential avenues for future work, thereby concluding this thesis.

Chapter 2

Background

In this chapter, I introduce the necessary background for Model-Driven Engineering. In Section 2.2, I present the activity of metamodeling and the involved artifacts. Section 2.3 discusses the automation task related to the artifacts presented in Section 2.2, followed by a presentation of the evolution and co-evolution concepts in the context of Model-Driven Engineering. I finish this chapter with few main information about Large Language Models.

2.1 Model-Driven Engineering

To develop a software, a list of specifications is given to the developers to code the final product. This approach can work in the case of small projects. When the complexity of the software increases, more efficient approaches must be adopted. Model-Driven Engineering has proven its efficiency comparing to other engineering disciplines in developing hyper-complex systems [21].

Model-Driven Engineering (MDE) is the systematic use of models as primary artifacts during a software engineering process. The usage of models allows more abstraction that helps in managing complexity. The first appearance of MDE-like approaches started in the 80's [22]. Till today, MDE is still adopted and a lot of work is being done in both academia and industry [8], [23]–[25]. MDE includes various Model-Driven approaches to software development, including Model-Driven Architecture, Domain-Specific Modeling and Model-Integrated Computing [26].

The goal of MDE is to improve productivity, quality, and maintainability by leveraging high level abstractions throughout the development process. MDE process includes many activities: metamodeling, model verification, code generation, model transformations, implementation, testing, and documentation. The metamodeling phase implied the experts of the domain who focus on the major key aspects of the problem rather than being concerned about the underlying programming language and the implementation. Moreover, it aims to improve communication between multi-disciplinary collaborators [25].

The metamodel represents the main artifact in MDE, it is also a main concept in my work. There are many definitions of the concept "metamodel" that can be found in literature, from Stahl et al. [27]:

Definition 01: *A metamodel* describes concepts that can be used for modeling the model (i.e. in the instances of the metamodel).

Definition 02: *Metamodels* are models that make statements about modeling. More precisely, a metamodel describes the possible structure of models in an abstract way, it defines the constructs of a modeling language and their relationships.

Definition 03: *A metamodel* defines the abstract syntax and the static semantics of a modeling language. Analogously, like a written program instance (e.g., in c or java, etc.) conforms to a grammar, a model instance conforms to a metamodel.

Seidewitz et al. [28] give another commonly used definition of *metamodels* in MDE:

Definition 04: *A metamodel* is a specification model for a class of systems under study where each system under study in the class is itself a valid model expressed in a certain modeling language.

2.2 Metamodeling

Metamodeling is the process of metamodel creation. Metamodeling is done thanks to metamodeling languages (that is in turn described by a meta-metamodel) as illustrated in Figure 2.1. Metamodeling must gather the whole knowledge that is required to define, precise, and deal with MDE challenges in its different tasks [25] related to other artifacts shown in Figure 2.2. The main metamodeling-related tasks are:

— The construction of metamodel itself: it describes the abstract syntax of target software languages or solution system.

— Model validation: models are validated against the constraints defined in the metamodel.

— Model-to-model transformations: such transformations are defined as mapping rules between source model and a target model .

— Code generation: it consists of automatically producing source code from models, bridging the gap between high-level abstractions and executable software.

— Tool integration: based on the metamodel, modeling tools can be adapted to the respective domain.

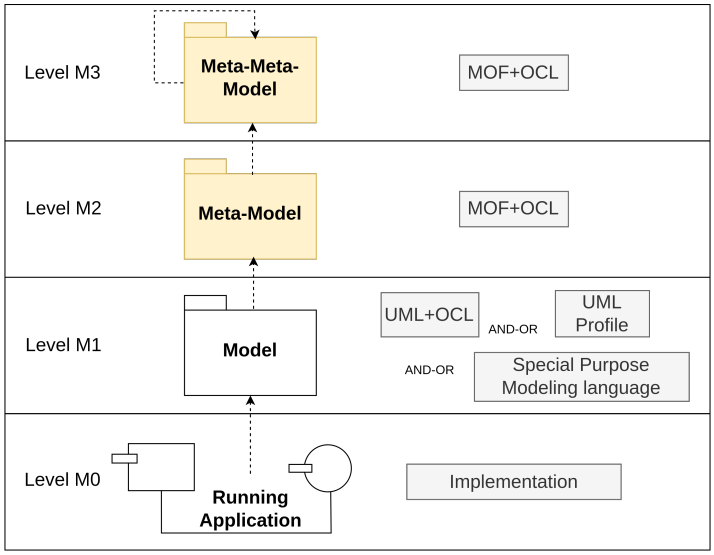


Figure 2.1 – MDA's Modeling Level Hierarchy

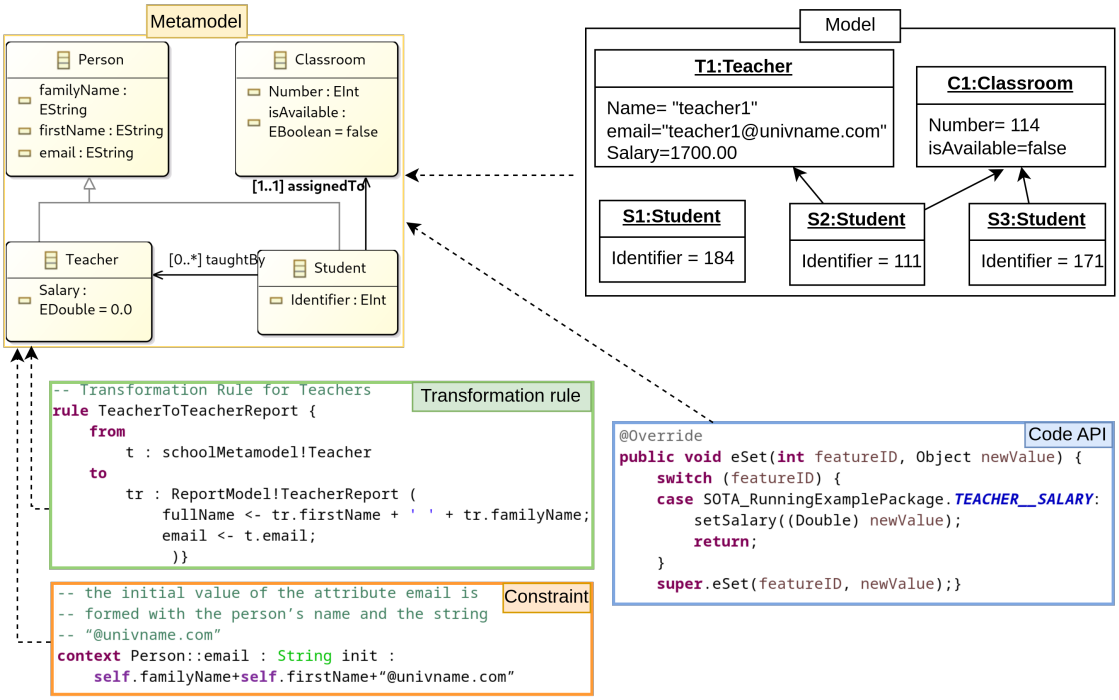


Figure 2.2 – MDE Ecosystem

Metamodeling languages are classified into two categories, namely linguistic and ontological [30]. Linguistic metamodeling represents a way for defining modeling languages and their primitives (e.g., Object, Class, MetaClass) on the layer of a metamodel. Ontological metamodeling aims to represent domain knowledge accurately. It is concerned with semantics and meaning, e.g., OWL [1]. Linguistic metamodeling aims to define a language for creating models. it is concerned with syntax and structure. I can use a different classification by purpose: General Purpose Modeling Languages and Domain-Specific Modeling Languages [31]. General Purpose Modeling Languages as for example : UML and its variants, generic metamodeling frameworks, such as MOF [2], and Ecore [3]. As examples of DSLs, I cite sysML and EXPRESS DSL [25]. As other toolings in MDE, there are language workbenches that are used for language creation, such as Xtext, MetaEdit+ [25].

Particularly, Domain-Specific Languages (DSLs) represent an essential application of metamodels in practice. A DSL is a specialized language tailored to the needs and constructs of a specific domain, enabling domain experts to express solutions at a higher level of abstraction without delving into general-purpose programming complexities. These languages are typically defined by a metamodel that establishes their syntax and semantics, ensuring consistency and precision. By leveraging metamodeling to create DSLs, engineers can align software artifacts closely with domain-specific requirements, enhancing clarity, maintainability, and automation potential in the software development process [29].

In the language modeling ecosystem, other artifacts are created by the mean of the metamodel. By definition, a model is an instance of a metamodel, which means that the metamodel defines the concepts with which a model can be created. The created models can also be validated through a set of constraints to check the models' correctness. Constraints are written in Object Constraint Language. They precise specifications on the model that cannot be expressed by diagrammatic notation. In order to save effort and avoid errors, models transformation is one of the common automated tasks in Model-driven engineering. Model transformation are expressed in Transformation Languages for example, ATL). A transformation consists of a set of rules that map the source metamodel elements to the metamodel target's elements. All of these artifacts have their specific tools and represent an important topic of research in MDE.

1. https://www.w3.org/TR/owl-features/
2. https://www.omg.org/mof/
3. https://eclipse.dev/modeling/emft/search/concepts/subtopic.html

2.3 Automation in the MDE ecosystem

Automation plays a pivotal role within the MDE ecosystem. It is considered as one of the most important advantages of MDE. This section explores the significance of automation in MDE, particularly in the code generation activity and during the evolution of the metamodel cornerstone artifact.

2.3.1 Code Generation

One of the most important activities in MDE is the code generation activity. It is recurrent, and its automation enhances the productivity and the cost. For example, Eclipse Modeling Framework built-in code generator allows to generated a java API from an Ecore metamodel. The generated code API structure and technical choices are done to fit Java programming language and Model-Driven Engineering abstraction standards and principles (e.g., each metaclass is used to generate an interface and concrete implementation class that extends the generated interface, pattern observer). The annotation *@generated* is used to mark generated interfaces, classes, methods, and fields. This annotation can be used to differentiate the generated code from the manually written one.

In Eclipse Modeling Framework, two model resources (files) are manipulated: the .ecore file that contains XMI serialization of the Ecore model and the .genmodel for the serialized generator model. The Ecore file is the document that contains the metamodeled main concepts that are used in code generation process.

2.3.2 Software evolution

From operating systems to mobile apps, software is the cornerstone of modern innovation. However, software does not remain static, it evolves over time to meet new demands,to address challenges, and to incorporate advancements. Here some numbers showing how much a software can evolve. Let's take Eclipse Modeling Framework (EMF) as a first example from MDE context. EMF [4] has 65 releases, with 10374 commits, and more than 823k lines of code. Another example from MDE is UML [5] with 2806 commits and more than 716k lines of code. The third example is Linux kernel [6] with 1324878 commits,

4. `https://github.com/eclipse-emf/org.eclipse.emf`
5. `https://github.com/eclipse-uml2/uml2`
6. `https://github.com/torvalds/linux`

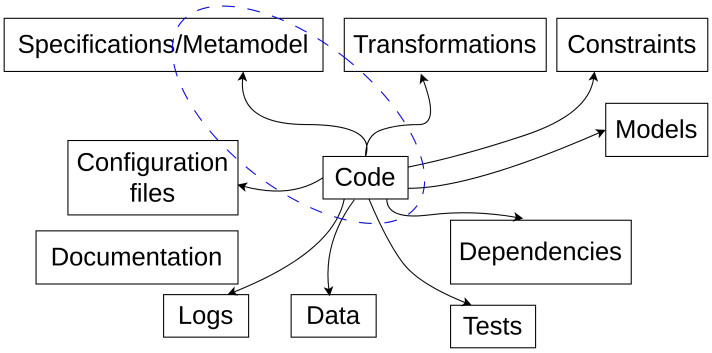


Figure 2.3 – software artifacts

and 40 million lines of code. Then chromium [7] with 1524157 commits and more than 32 million lines of code. The last example is cpython [8] with 124936 commits, more than 350k lines of C code and 600k lines of python code. During the software development process, software artifacts are meant to be changed, due to many reasons: client requirements and domain specifications, software maintenance, or bug correction. The evolution can impact one artifact ore more in the software ecosystem (Figure 2.3). Like any other software system, modeling languages are the subject of an inevitable evolution, during their process of building, multiple versions are developed, tested, and adapted until a stable version is reached.

Different types of evolution are categorized depending on the impact and purpose of the applied modifications [32], [33]:

— Corrective: aims to correct discovered problems and inconsistencies, such as processing failures, performance failures, or implementation failures by applying a set of reactive modifications of a software product.

— Adaptive: in case of changing environment, such as changes in data environment or processing environment, this evolution aims to keep a software product usable.

— Perfective: this evolution aims to improve functionalities, to enhance the performance, reliability, or to increase the maintainability of a software.

The term *Evolution* can be refined as the literature presents various related terms like: Maintenance, Refactoring, and **Co-evolution**, which are different types of modifications

7. `https://github.com/chromium/chromium.git`
8. `https://github.com/python/cpython`

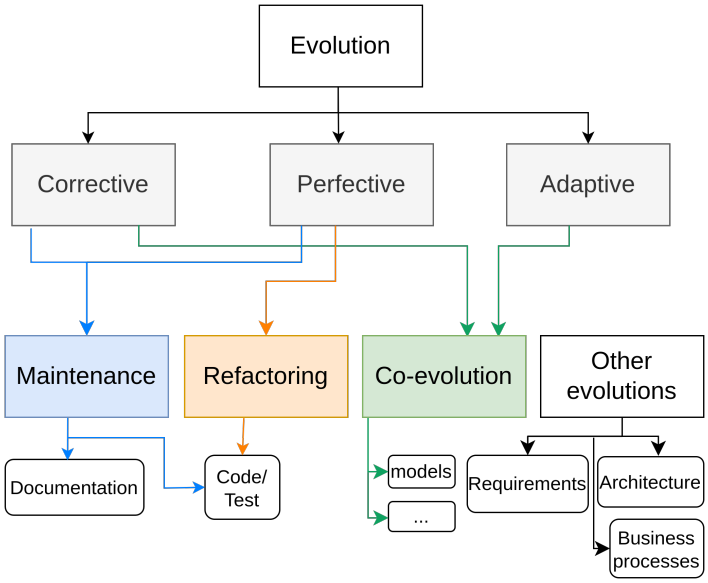


Figure 2.4 – Evolution related terminology

that could be applied on a software. There is no clear definition for each of them, however, in Figure 2.4 I describe the relationship between these terms by intent and by artifacts on which the task is applied.

Evolution: any adaptation that occurs in the software in response to new requirements. These requirements are the consequence of the past experience of the users that feeds the developers' learning. [34].

Maintenance: It is modifying a software product after delivery to correct faults, to improve performance, or to adapt the product to a changing environment [35].

Refactoring: It is an oriented object term, that means behavior-preserving modifications of software, to make it easier to understand and to change or to make it less susceptible to errors when future changes are introduced [36].

Co-evolution: It consists of the process of adapting and correcting a set of artifacts A_1, A_2, ...A_N in response to the evolution of an artifact B on which A_1, A_2, ...A_N strongly depend, for example the co-evolution of models with the evolving metamodel as used in Kessentini et al. paper [37], and the co-evolution of API/client as used in Eilertsen et al. paper [38]. The term "Coupled evolution" was also mentionned in the work of Herrmannsdoerfer et al. in the context of metamodel and model co-evolution [39].

2.4 Large Language Models

In this section, I outline two fundamental concepts related to Large Language Models, namely Transformer models and prompt engineering. I also give a short historical view to understand the added value of using LLMs in my third contribution (cf. Chapter 6).

2.4.1 From Transformer Models to Large Language Models

Machine learning field introduced statistical models that learn from input data. As an improvement of traditional machine learning approaches, neural networks brought breakthroughs in image and speech recognition, inspiring their application to Natural Language Processing (NLP). A markup point in the history of machine learning is The Attention Mechanism [40]. It allows models to focus on relevant parts of input sequences, significantly enhancing translation and comprehension tasks. This mechanism allows the model to weigh the importance of all words in a sequence simultaneously, leading to: 1) Parallel processing for a faster computation, and 2) Better handling of long-range dependencies. After neural networks and Attention mechanism application in NLP, the transformer models appearance first in 2018, marked by the emergence of BERT and its variations [41].

BERT (Bidirectional Encoder Representations from Transformers) is an encoder-only transformer model owned by Google [42]. Regarding data corpus, BERT is pre-trained from Wikipedia and Google's BooksCorpus. Another remarkable model is **BART** (Bidirectional and Auto-Regressive Transformers). It was introduced by Lewis et al. [43]. Unlike BERT's encoder-only design, BART's architecture is a sequence-to-sequence model, featuring both encoder and decoder components. In 2018, OpenAPI has launched another large scale pre-trained transformer model: GPT thus marking the beginning the era of Large Language Models (LLMs), then the important **GPT-3** appearance in 2020 [44].

Note that in the field of LLMs, new approaches are being developed at a rapid pace. Thus, rather then latest models, I selected models that have marked a turning point in the history of LLMs: BERT, BART, and GPT to include in this short LLM background. No global and detailed comparison between GPT, BERT, and BART is so far established. However, some works have conducted targeted comparisons in specific context. For example, Yokoyama et al. compare between Chatgpt and BERT in security bug identification topic [45]. Table 2.1 summarizes the key differences between BERT, BART, and GPT-3 depending on: their architecture, learning data, number of parameters,

Table 2.1 – Comparison of BERT, BART, and GPT-3

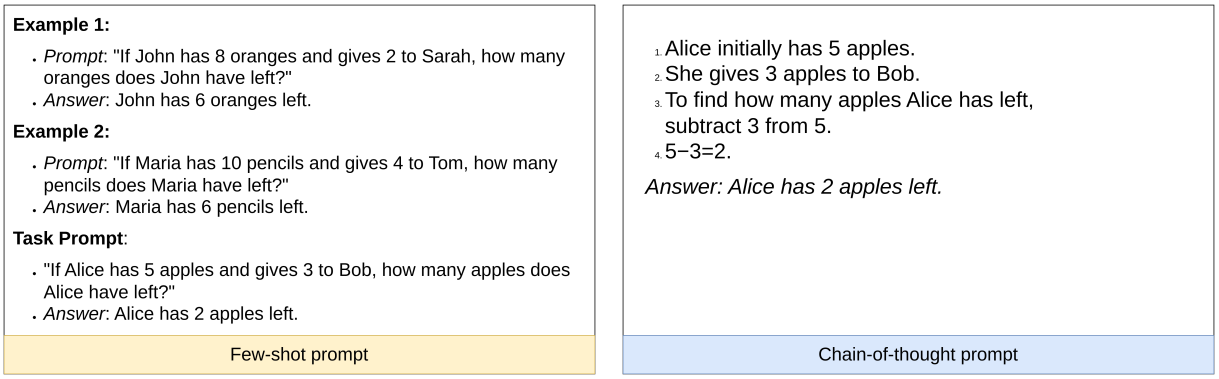
Functionality	BERT	BART	GPT-3
Architecture	Bidirectional encoder-only model that considers both left and right contexts when making predictions.	Encoder-decoder model, Bidirectional encoder with left-to-right autoregressive decoder it considers both left and right contexts when making predictions.	The autoregressive model generates text by predicting the next word in a sequence based on the preceding words.
Data	Learning on data from sources such as English Wikipedia and BookCorpus 11038 books).	Learning from a combination of books and Wikipedia data.	extremely large corpus of English text data extracted from millions of web pages and then fine-tuned using Reinforcement Learning from Human Feedback (RLHF).
Number of parameters	$BERT_{base}$ 110 m parameters $BERT_{large}$ 340 million parameters, a relatively smaller number than GPT-3.	$BART_{base}$ contains 139 million parameters and $BART_{large}$ contains around 400 million parameters.	175 billion parameters, far more than any other language model.
Performance speed?	faster than BART and GPT-3 .	Decoding step and the number of parameters make it slower than BERT.	the slowest one, because of the number of parameters, size of data, and the complex architecture.
Application (domain)	Top performance on a range of tasks, including text classification. Requires further fine-tuning and learning to adapt to new domains and tasks.	particularly effective when fine-tuned for downstream NLP tasks, especially text generation.	It has shown remarkable performance on a wide range of natural language processing tasks. Can be generalized to new domains thanks to learning in a few steps, and adapted to new tasks thanks to transfer learning.
Origin organisation	Google AI	Facebook AI	OpenAI

performance speed, application domain in which the model is most efficient, and the respective organizations that created them [41], [43].

2.4.2 Prompt Engineering

Definition – Prompt Engineering. It consists of techniques that are used to converse with LLMs **velasquez2023prompt**. As a subdomain of LLMs field, it represents an active and rapidly evolving area of study. Frieder et al. confirm the impact of having a correct prompt structure in the capability of LLMs' problems resolution [46]. Although GPT-3 may have contributed to these prompt engineering techniques' popularization [47], other LLMs are capable to adopt these techniques, such as PaLM [48] and Chinchilla [49].

Here I define two popular categories of prompt-engineering techniques: zero-shot/few-



(a) Few-shot prompt example (b) Chain-of-thought prompt example

Figure 2.5 – Few-shot vs Chain-of thought example

shot prompting and Chain-of-Thought reasoning:

— Few-shot/zero-shot prompting: in few-shot prompting, the prompt is enriched with one or more examples of the task in the prefix before being asked to perform the task on new data. These examples serve as a way to prime the model to produce the desired output. In zero-shot prompting, no example is provided [50], [51].

— Chain-of-Thought reasoning: enables complex reasoning capabilities through intermediate reasoning steps. it can be combined with few-shot prompting to get better results on more complex tasks that require reasoning before responding [52].

Note that other techniques can be found in Liu et al. systematic survey of prompting methods in [51], and in this prompting guide website [9].

Let's take an example for the prompt "If Alice has 5 apples and gives 3 to Bob, how many apples does Alice have left?". Figure 2.5a shows the prompt content in the case of few-shot prompting, and Figure 2.5b shows the chain-of thought reasoning for the same prompt.

2.5 Conclusion

In this chapter, I provided the essential background for Model-Driven Engineering, covering key aspects such as metamodeling and its related artifacts, automation tasks, and the concept of co-evolution with the different concepts related to evolution. Additionally, an overview of Large Language Models was presented to highlight their relevance in this

9. `https://www.promptingguide.ai/techniques`

thesis. This foundation establishes the necessary understanding for the exploration of following chapters.

Chapter 3

State Of The Art

In this chapter, I present an overview of what has been done in the field of Model-Driven Engineering in the context of code co-evolution. I split this overview into five parts. In section 3.1, I present the metamodel change detection approaches. Section 3.2 presents the co-evolution of model, transformations, constraints with evolving metamodel. In section 3.3, I discuss code co-evolution and relevant literature about API-client evolution, and language evolution. In Section 3.4, I browse related work to checking the behavioral correctness of code co-evolution. Section 3.5, presents an overview of the use of LLMs in related MDE and SLE tasks. I finish this chapter with a discussion focused on limitations and research gap in Section 3.6.

3.1 Metamodel change detection

One of the intrinsic properties of software artifacts is its continuous evolution [53]. Like any software artifact, metamodels are meant to evolve to meet the represented domain. In this thesis, the context is triggered by the metamodel evolution, that's why I find essential to understand this evolution in detail. A lot of work has been done on metamodel diffing. Detection approaches can be classified into two main categories: online [1] detection approaches, and offline [2] detection approaches. This classification can be refined using some factors as detailed by Hebig et al. [54]: automation degree, types of detected changes, and considered issues (overlap, indefinite length, hidden changes, order of changes, and undo operations) [54].

Furthermore, many of them classified the detected changes based on their impact on the treated artifact (e.g., models, constraint, transformation, and code). In Table 3.1, I put the largest set of changes types that I found in literature [55]. Later in Section 4, I specify the treated subset of these changes.

1. Offline approaches perform detection after the metamodel has been evolved.
2. Online approaches perform instant detection for each change during the metamodel evolution

Type	group	Change name
Atomic changes	Structural Primitives	Create Package, Delete Package, Create Class, Delete Class, Create Attribute, Create Reference, Delete Feature,change type, Create Opposite Ref., Delete Opposite Ref., Create Data Type, Delete Data Type, Create Enum, Delete Enum, Create Literal,Merge Literal
	Non Structural primitives	Rename, Change Package,Make Class Abstract, Drop Class Abstract, Add Super Type, Remove Super Type, Make Attr. Identifier, Drop Attr. Identifier, Make Ref. Composite, Switch Ref. Composite, Make Ref. Opposite, Drop Ref. Opposite
Complex changes	Specialization / Generalization Operators	Generalize Attribute, Specialize Attribute, Generalize Reference, Specialize Reference, Specialize Composite Ref. Generalize Super Type, Specialize Super Type
	Inheritance Operators	Pull up Feature, Push down Feature, Extract Super Class, Inline Super Class, Fold Super Class, Unfold Super Class, Extract Sub Classn Inline Sub Class
	Delegation Operators	Extract Class, Inline Class, Fold Class, Unfold Class, Move Feature over Ref., Collect Feature over Ref.
	Replacement Operators	Subclasses to Enum., Enum. to Subclasses, Reference to Class, Class to Reference, Inheritance to Delegation, Delegation to Inheritance, Reference to Identifier, Identifier to Reference
	Merge / Split Operators Merge	Merge Features, Split Reference by Type, Merge Classes, Split Class, Merge Enumerations

Table 3.1 – Catalog of model operators

In Model-Driven Engineering, depending on their impact on model instances, metamodel changes can be divided into three categories [56]:

— Non-breaking changes that occur in the metamodel but do not break other artifacts of lower abstraction level.

— Breaking and resolvable changes break the conformance of existing data, although they can be automatically adapted.

— Breaking and unsolvable changes break the conformance of existing data, that cannot

be automatically adapted, and require user intervention to be resolved.

In API evolution similar context, API changes can be classified as non-breaking API changes or breaking API changes. A non-breaking change is backward compatible. This kind of changes aims to extend the functionalities or fix errors. A breaking change is not backward compatible. In this case, client code calling the evolved API by a breaking change fails to compile or may behave differently at runtime [57].

In literature, two types of evolution changes are considered when evolving a metamodel: *atomic* and *complex* changes [54], [58]. Atomic changes are additions, removals, and updates of a metamodel element. Complex changes consist of a sequence of atomic changes combined together [59], [60]. For example, move property is a complex change. Where a property is moved from a source class to a target class. This complex change is composed of two atomic changes: delete property and add property [58]. Many approaches exist to detect changes between two metamodel versions [59], [61]–[66].

Demuth et al. [67], Herrmannsdoerfer et al. [39], Khelladi et al. [68] are online approaches that take into consideration undo operations while the metamodel evolution. In contrast, other approaches [59], [62]–[64], [69]–[74] adopt offline detection method. However, offline detection may cause an order issue, particularly in [62], [64], [69]–[74]. Notably, none of these approaches consider hidden changes, with the exception of Vermolen et al. [57].

Regarding to the automation degree of the approaches, Herrmannsdoerfer et al. [39] and Williams et al. [62] are manual methods. Di Ruscio et al. [69], Vermolen et al. [59], and Khelladi et al. [68] are all semi-automatic that require user decision to select final output changes. Demuth et al. [67]; Levendovszky [70],Cicchetti et al. [63], Garces et al. [71], Langer et al. [64], Garcia [72], Xing et al. [73], Moghadam et al. [74] are full automatic approaches.

Moghadam [74], Vermolen [59] and [68] take into consideration indefinite length of a complex change. Khelladi et al. [68] take into consideration overlap issue.

All these approaches propose structural changes detection: additions, deletions, or modifications to classes, attributes, associations, or inheritance structures [75]. When a metamodel is defined, the definition of the structural side of the representation of a domain is given. The behavioral side is given through constraints [76]. In this thesis, I focus only on the structural metamodel changes, any change occurring on behavioral aspects of a language, at metamodel level, is out of the scope of this dissertation. After studying the literature change detection approaches of metamodel, relatively to the challenges that I discussed in the challenges section 1.2, I found that Khelladi et al. [68] handle a large set

of changes. Furthermore, their approach handle all the issues that I mentioned. It is a semi-automatic approach but this adds a trust value to the approach because automatic approach may have order or overlap issues.

Another reason to choose Khelladi et al. [68] approach is their output representation and vocabulary. It required a minimum effort of adaptation because I use the same changes representation that I describe in Section 4.2.2.

3.2 Co-evolution of models, constraints, and transformation

In this section, I will present an overview of the existing work about these artifacts co-evolution. Note that the comparison, advantages, and drawbacks of the presented approaches is out of the scope of this dissertation.

3.2.1 Metamodel and model co-evolution

Due to metamodel evolution, instance models become no conformant. A set of resolutions are applied to co-evolve the models to gain again the conformity to the metamodel. Two strategies to evolve models due to metamodel evolution. In the first strategy, the metamodel is the artifact to be adapted in a way that the old models can still be used with the evolved modeling language without adapting the models [39]. This approach suggest the resilience of the models. The second strategy adapts the metamodel in a possibly breaking manner for the models, that must be adapted by transforming them into a new version that conforms to the adapted metamodel.

The co-evolution between metamodel and models can be processed manually, but it requires a huge expertise, and when the number of models to co-evolve increases, manual co-evolution becomes hard task. Most of automatic and semi-automatic co-evolution approaches use automatic or manual diffing metamodel approaches in their solutions. Model co-evolution approaches that exist can be categorized into five categories [77]. The first category is Resolution Strategy Languages that specify in a transformation language how to update the model given the list of metamodel changes [78]–[84]. The category Resolution Strategy Generation groups approaches that generate full or partial resolution for each metamodel change [71], [85]–[88].

The third group of Predefined Resolution Strategies contains approaches that provide automation, when it is possible, by applying predefined resolution strategies [89]–[97].

Some of these approaches require user intervention to make decision on the selected operation to adapt the model. The fourth category of Resolution Strategy Learning that adopts machine learning algorithm to select the resolution strategy for metamodel changes.[98].

The fifth and last category is called Constrained Model Search approaches. It groups approaches that do not use the metamodel changes, but use the original model and the new metamodel to apply a constrained-based search of valid model variants [99]–[101]. Other approaches consider the model co-evolution problem as an optimization one that does not need the list of changes of the metamodel [102]–[104].

Depending on whether a user intervention is needed or not to apply predefined or generated resolutions, there are automatic and semi-automatic approaches. In one hand, Di Ruscio et al. [105], Levendovszky et al. [70], Anguel et al. [88], [98], Cicchetti et al. [106], and van den Brand et al. [94], are automatic model co-evolution approaches. In another hand, there are approaches that I found in literature that are semi-automatic: de Geestwe et al. [86], Garcès et al. [107], Wittern et al. [97] (their approach is automatic for atomic changes and manual for complex changes).

Related to the previous section 3.1, I found that only Demuth et al. [99] studied the impact of metamodel evolution on models. Their consistent change propagation focuses on maintaining consistency between artifacts using constraints (metamodel itself and existing models). Later, the impacted constraints are used to propose repairs, then manual intervention is needed to select and apply these repairs. Where the other model co-evolution approaches of COPE [39], Wachsmuth et al. [92], and Kessentini et al. [104], [108] do not explicitly study the impact of metamodel evolution on models. Cicchetti et al. [106] categorize metamodel modifications into additive, subtractive, and updative. Their approach starts by generating a difference model, then a transformation model to co-evolve models, without a step dedicated impact analysis. Garcès et al. [71] computes equivalences and differences between any pair of metamodels, simple and complex changes. These equivalences and differences are then represented as matching model. In the second step, the matching model is translated into an adaptation transformation by using a Higher-Order Transformation (HOT) that is later executed. A matching model is used to generate a transformation model but no explicit analysis for the impact of metamodel evolution on the models is provided.

3.2.2 Metamodel and constraints co-evolution

Another artifact that depends on the metamodel and needs to be adapted as the metamodel evolves is constraints. Constraints co-evolutions that exist in literature may be online [3] or offline [4]. Every approach has its own co-evolution mechanism that treats specific types of metamodel changes and has its automation degree. Demuth et al. [109] propose a template-based of the predefined structure of the updated constraint taking into consideration few change types. Markovich et al. [110] propose refactoring rules that depend on the impact of UML class diagram evolution on the constraints. In this approach, the user selects the refactoring rule to be applied on the model then on the depending constraints. Another constraints co-evolution approach is METAEVOL proposed by Hassam et al. [111]. METAEVOL is a tool based on a transformation language. Kusel et al. [112] propose a solution for the co-evolution of the constraint body and do not include its context that may need to be co-evolved also. Cabot et al. [113] treat OCL constraints co-evolution due to metamodel deletion change. Khelladi et al. [114] propose an approach that records the metamodel atomic and complex changes in a chronological order, then apply one or many resolutions to co-evolve the constraints. Batot et al. [115] tackle the constraint co-evolution problem as a multi-objective optimization problem. They apply heuristic-based recommendation approach that does not use a predefined set of transformation rules to co-evolve the constraints.

Cherfa et al. [116] provide an assistance to developers working on OCL constraint co-evolution, by focusing on the structures in the metamodel that potentially cause problems and which ones need new OCL constraints (after the co-evolution). Their approach does no explicitly process an impact analysis.

Regarding the impact analysis of the metamodel evolution, Kusel et al. [112] study this impact on OCL expressions. They distinguished between breaking and non-breaking impact. Moreover, they divided the changes into three groups: constructive changes which are non-breaking where destructive changes and updative changes which are considered breaking changes if they have at least one breaking impact case on the constraints. Furthermore, their study includes every possible case and the corresponding resolution. This paper [112] proposes syntactic co-evolution that can be checked by a compiler, where the semantic co-evolution correctness is checked through Pattern-based formal specification Modeling Language for Model Transformations (PaMoMo) [117]. The correctness checking process

3. Online approaches perform instant co-evolution for each change during the metamodel evolution
4. Offline approaches perform co-evolution after the metamodel has been evolved.

uses a set of Pamomo specifications (input models) to be verified before and after OCL expressions' co-evolution.

In their co-evolution approach, Batot et al. [118] first register metamodel elements that were deleted, added, or had their multiplicity changed between the two versions. Then,they apply an NSGA-II heuristic algorithm to satisfy the objective functions. However, no additional correctness checking are performed. Finally, a syntactic comparison is processed to determine if a candidate constraint is the same as the expected one. Notably, the impact of metamodel evolution on OCL constraints is not explicitly analyzed.

Hassam et al. [111] proceed by a partial impact analysis, through a table that contains constraints context that is linked with the involved elements of the metamodel that is evolved. The designer is then, responsible to check the validity of the constraint.

Khelladi et al. [114] associate each evolved metamodel element to impacted constraint (context and body) and precise the impacted constraint astnodes. However, no further correctness checking of their output.

Cabot et al. [113] as an impact analysis, identify the elements to be deleted (those selected by the user plus all the elements affected by them).

Regarding the automation degree of the approaches above, Demuth et al., Markovitch et al., and Cabot et al. [109], [110], [113] approaches are fully automatic while Hassam, kusel and Khelladi, and Batot [111], [112], [114], [115] propose semi-automatic approaches since the user must select from the recommended output constraints.

3.2.3 Metamodel and transformations co-evolution

Almost all the existing transformation approaches that I find in literature have the same strategy. This strategy consists of evolving the impacted parts in the transformations in a semi-automatic manner requiring a human intervention. Garcia et al. [72] proposed an ATL rule-based approach. This approach only guarantees that the evolved transformations are syntactically correct, but does not process any other correctness checking, including semantic correctness. Kusel et al. [119] explained explicitly that his approach verifies semantic correctness through a set of properties expressed in PaMoMo language [117] as a kind of regression testing.

The approach of Khelladi et al. [120] covers the largest set of existing resolutions. In their change propagation-based approach, they allow to compose existing resolutions into a new one. This approach [120] does not process any semantic verification of the transformation co-evolution alongside the comparison with their ground of truth. Ruscio

et al. [121] with their EMFMigraite tool, estimate first the cost of the co-evolution to decide about the co-evolution. EMFMigrate further explores the variability and common aspects of the co-evolution related to different type of artifacts. Moreover, they allow developers to manually replace or refine a resolution. All these approaches propose a unique resolution to co-evolve each transformation. Mendez et al. [122] study the relation between the metamodel and the transformations. It detects the transformations' inconsistencies after the metamodel co-evolution. then, depending on the impacting metamodel change, a resolution is suggested. Garcés et al. [107] propose a solution based on adaptation chains via composition functions depending on the impacting metamodel changes. Kessentini et al. [123] followed a different approach that does not use the changes of the metamodel as input and does not process an impact analysis, but rather uses a search-based approach that relied on multi-objective heuristic algorithm NSGA-II. This approach has an objective function to minimize the number of errors of non-conformance between the metamodels and transformations. These errors can be statically detected by static semantic constraints. However, no semantic verification of the co-evolution correctness is processed.

Mendez et al. [122], Garcés et al. [107], and Ruscio et al. [121] do not include any impact analysis of metamodel evolution on the transformations to be co-evolved. However, Garcès et al. [107] explains the impact of the evolution of the metamodel on the transformations through a motivating example but not an explicit study independent from the example. I found that only Mendez et al. [122] and Khelladi et al. [120] dedicated a detailed part in their approaches for the analysis of the impact of metamodel evolution on the transformations.

Even though I consider the co-evolution in a larger scope in Model-Driven Engineering, my main focus is given to the co-evolution of the code that I detail in next section.

3.3 Code co-evolution

I divided the related work to code co-evolution into four (4) main categories : 1) Metamodel and code co-evolution, 2) API and client code co-evolution, 3) Automatic Program repair, and 4) Consistency checking.

3.3.1 Metamodel and code co-evolution

Co-evolution of code is distinguished from the co-evolution of other artifacts. This distinction is due to the fact that the code and other artifacts (models, constraints, transformations) are on different levels of abstraction. In fact, the models and constraints are on closer level of abstraction of the metamodel, where each metamodel element is directly referenced/present in the depending artifacts. However, the code is on a lower level of abstraction where each metamodel element has different representation in the code. Thus *one* change in a metamodel element will affect n different code elements, in contrast to a *one* to *one* impact relationship between metamodel elements and models, constraints and transformation elements [39], [71], [92], [103], [106]–[108], [114], [118]–[120], [123]–[126].

Yu et al. [10] propose to co-evolve the metamodels and the generated API in both directions. However, they do not co-evolve the code on top of this generated API. Khelladi et al. [11] propose an approach that propagates metamodel changes in the code as co-evolution mechanism. Nonetheless, it is based on static analysis to detect the impacts and not on the actual errors that appear from the compilation of the code after the metamodel evolution. It further applies a semi-automatic co-evolution requiring developers' intervention, and without checking behavioral correctness with tests or any comparison to a baseline.

3.3.2 API and client code co-evolution

Existing approaches for code migration, whether library code or API calls, are related to our work. Henkel et al. [127] propose an approach whose implementation is called CatchUp!. It captures refactoring actions of the library and replays them on the code to migrate. However, they support limited types of changes as renaming, moving, and class type changes.

Nguyen et al. [128] also propose an approach that guides developers in adapting code by learning adaptation patterns from previously migrated code. Similarly, Dagenais et al. [129]–[131] also use a recommendation mechanism of code changes by mining them from previously migrated code. Anderson et al. [132] proposed to migrate drivers in response to evolutions in Linux internal libraries. It identifies common changes made in a set of files to extract a generic patch that can be reused on other code parts. Gerasimou et al. [133] extract a set of mapping rules and apply code-based transformations to update its clients.

Zaitsev et al. [134] present a survey about library evolution, that involves developers from two industrial companies: Arolla and Berger-Levrault in a hand, and Pharo as an Open-Source Community in another hand. In this survey, the study was conducted in both perspectives: client side and library side. Kula et al. [135] studied from library side, the impact of refactoring activities on evolving client-used APIs. Other works focused on the client side and how do client applications' developers react to the evolution of the libraries they depend on [136]–[141]. Jezek et al. [142] treat both client and library perspectives. They studied the compatibility aspect of the APIs and the impacts of the library evolution on the programs using it. Further detail about the supporting library update can be found in the PhD thesis of Zaitsev Oleksandr [143]. Shaikh et al. [144] studied Behavioral Backward Incompatibilities. In their paper, they process a cross-version regression testing to understand the behavioral changes of APIs during evolution of software libraries.

Other migration approaches [145]–[147] rely on pre-collected examples to learn how to evolve the additional client code. Xu et al. [148] instead of learning from code examples, they construct a database of edits to use during clients' migration.

Fazzini et al. [146] propose to check the correctness of the code migration using differential testing but it still needs previous example-based learning to update the client code. Zhong et al. [149] proposes "LibCatch" to co-evolve client code to APIs evolution by reducing the compilation errors. They do not consider the API changes to correctly propagate them to the code, which may lead to only eliminating the code errors while they could be incorrect resolutions. Moreover, they do not use any mechanism to check the behavioral correctness of the code co-evolution and with not comparison to a ground-truth.

Di Rocco et al. [150] propose DeepMig, a dual purpose tool for both third party libraries and API migration. DeepMig is base on a transformer machine learning model. It starts by mining projects' history and different changes in order to provide recommendations to migrate both libraries and code. In their evaluation, they do not check the conservation of the behavioral correctness of the code.

3.3.3 Automatic Program Repair

Even though Automatic Program Repair (APR) approaches are not considered as co-evolution approaches, I include APR related work in this section because APR is an automatic code modification, whose correctness needs to be checked, which is related to the challenges cited in the introduction 1. Extensive state of the art exists on APR field [151]–[154]. However, they do not repair code errors, but rather bugs that are found due to

failing tests (e.g., Meng at al. [155]). They could be used as a next step after o co-evolution. Xia et al. [156] conducted a study on the application of Pretrained Language Models (PLMs) including both generative and infilling models on APR. This study investigated the ability of PLMs in generating correct patches and its performance in ranking these patches, in addition to its performance in scaling. Claire et al. [151] give a review article about Automatic Program Repair. Their paper presents an overview of the APR techniques that has as input a buggy program and most of them use test suites for correctness checking.

Ruan et al. [157] propose a co-evolutionary-based approach for APR. This means that they aim to evolves two populations simultaneously: a set of patches and a test suite. They implemented their workflow as a tool called EVOREPAIR as an extension of EvoSuite [158]. Xia et al. [159] propose ChatRepair which is a fully automated conversation-driven tool. ChatRepair leverages ChatGPT to perform repair. This tool uses previously incorrect and plausible patches and test failure information as an immediate feedback to get better generated patches.

Chen et al. [160] propose LIANA wich is test-driven generate-and-validate program repair loop. It is based on repeatedly updating a statistical model by learning the features of the fix candidates. LIANA starts working using a given java program with a test suit that has at least one failing test.

3.3.4 Consistency checking

Close to code co-evolution, Riedl et al. [4] proposed an approach to detect inconsistencies between UML models and code. Kanakis et al. [5] showed that inconsistency information of model change and code error can help to resolve them in the code.Pham et al. [6] proposed an approach to synchronize architectural models and code with bidirectional mappings. Jongeling et al. [7] proposed an early approach for the consistency checking between system models and their implementations by focusing on recovering the traceability links between the models and the code. Jongeling et al. [8] later rely on the recovered traces to perform the consistency checking task. Zaheri et al. [9] also proposed to support the checking of the consistency-breaking updates between models and generated artifacts, including the code. However, some approaches [6]–[9] do not focus on co-evolving the code to repair the inconsistencies with the models.

3.3.5 Language evolution

Language evolution is related to various technological spaces [161]. Metamodels evolution [162] (Section 3.2), APIs' evolution [57] (Section 3.3.2), grammars' evolution in [163], schemas' evolution [164], [165], and ontologies' evolution [166].

There are two types of languages: Domain-Specific Languages (DSLs) and General Purpose Languages (GPLs).

DSLs are strictly coupled to the domain and its requirements/capabilities at the time in which the DSL is written. If the domain requirements and/or capabilities change, then the DSL could become inadequate to deal with the changed domain. Schuts et al. [167] incrementally changed a five year old DSL called Azurion that supports multiple hardware preserving its behavior. Initially these configurations were prefixed. After the evolution of the DSL, the configurations can be defined by the user. As DSLs evolve [168], [169], the presence of inter-DSL dependencies in a Model-Driven Systems Engineering (MDSE) ecosystem causes a ripple effect and increases costs of manual maintenance. Hence, an automatic approach is required to facilitate co-evolution of artifacts in MDSE ecosystems. Regarding domain-specific language evolution, works treated this topic from many aspects, and in different ecosystems and case studies [170]. On the one hand, there are open source ecosystems case studies like The Graphical Modeling Framework (GMF) [5] which is a widely used open source framework for the model-driven development of diagram editors implemented on top of the Eclipse Modeling Framework (EMF).

Herrmannsdoerfer et al. present a method to investigate the evolution of modeling languages, and hint at the possible effects of this evolution on the related language development artifacts [171].

On the other hand, there are industrial ecosystems case studies like CARM (Control Architecture Reference Model) which is an industrial ecosystem for ASML [6] which is the world's leading provider of complex lithography systems for the semiconductor industry [170].

Regarding General Purpose Languages that are simply programming languages, their evolution consists mainly of improvements of syntax and semantics or feature additions that allows to some extent the stability of the code and do not break it. The evolution of the programming languages is due to two types of causes: 1) External, such as hardware changes, the control of business needs, or the progress of scientific research. 2) Internal

5. https://eclipse.dev/modeling/gmp/?project=gmf-runtime
6. https://www.asml.com/en

for bug fixing, enhancing the grammar, or improving the verbosity of the language [172]. For instance, the introduction of the generics since Java 5 [7] as new feature exemplifies one of Java evolutions. Another example of an explicit semantic change in the division operation from Python 2 to Python 23. In Python 2, an expression such as 1/2 returns 0. However, 1.0/2 returns 0.5. In contrast, in Python 3, the division operator has a float return type and the result is 0.5, whether the division operators are ints or if one the them is a float.

Every change to a programming language API has the potential to affect programs written in this language. For example, Python 2 and 3 have major incompatibilities that leads to maintenance costs, particularly, due to Python's dynamic typing, it is difficult to locate some errors that can be found during program execution. It is worth noting that manually resolving the incompatibilities resulting from language evolution is a daunting task. Dietrich et al. [173] show that developers lack awareness of the often present limitations and possible incompatibilities which make code maintenance a hard task. Regarding programming languages evolution, many works were centered around change-impact analysis before proposing an adaptation approach [174]–[176].

Urma et al. [172] propose PytypemonitorInfer, a dynamic light-weight type inference tool for Python to automatically provide insights useful for migration about a Python program. Few works were done for specific languages evolution. For example, the expansion and evolution of the R programming language regarding linguistic understanding of human language [172]. Another example for Pharo programming language, where Hora et al. [137] analyzed in their empirical study the most important changes in Pharo API and their impact on different systems in Pharo ecosystem. Ochoa et al. [177] propose Breakbot, a tool to analyze the impact of the breaking changes of java libraries on their client code.

More insights about language evolution can be found in the dissertation of Raoul-Gabriel Urma [172].

3.4 Behavioral correctness of the co-evolution

In previous sections, I browsed different approaches for code co-evolution. Whether the adopted approach is manual, semi-automatic, or automatic, it is essential to make sure that this co-evolution was processed correctly.

7. https://blogs.oracle.com/javamagazine/post/understanding-java-generics-part-1-principles-and-fundamentals

To ensure the behavioral correctness of a given version of the code, we can follow different methods. The first one is manual and exhaustive debugging to ensure that the code acts as expected. This method can be tedious and error prone for relatively complex code. Another method which is widely exploited is Testing. Testing in software engineering is a large research area with different approaches and tools. To check the behavioral correctness of a program, we can use Unit Testing through good quality unit tests that pass. In the case of more critical systems (Medical devices systems, autopilot systems, avionics engineering..etc), it is primordial to use formal methods like model checking or theorem proving [178] which is very costly.

In the context of code evolution, I need to verify that the code changes did not alter its behavior. In more large perspective, I would like to check the impact of this evolution on the code, did it improve, kept, or alter the behavior of the code. From this point of view, I investigated the literature to explore the different approaches that are used for this purpose. In other words, I investigated the approaches that check the impact of a code's evolution on the behavioral correctness of the code, whatever the evolution type is.

In Automatic Program Repair(APR), after fault localization, the goal is to remove bugs. The correctness of this bug removal is often checked using test suit that passes. Ruan et al. [157] propose an approach to check the correctness of the generated repair patches. Their approach is an extension of EvoSuite [158], that has two outputs. First output is the repair patches of better quality and the second output is the tests that prove the veracity of the patches.

Qi et al. [179] also divide existing approaches for APR assessment into two main approaches on assessing patched program correctness: formal specifications and APR assessment metrics with test suits.

Liu et al. [180] state that in the literature, correctness is generally assessed manually by comparing the generated patches against the developer-provided patches. Moreover, the evaluation metrics of APR methods could be biased [180]. In their paper, they exploit the number of bugs for which a correct patch is generated. Other metrics as the number of successfully fixed bugs with patches that can pass all the given test cases. This metric can be biased when the generated patched pass all the tests but introduce other faults that are not covered by these tests [180]. This paper proposes metrics that limit the biases when assessing APR tools. The list of the used metrics:

• Upper bound Repair Performance metric indicates the patch generation limitations when referring to the exact bug-fixing positions obtained from the ground of truth developers'

patches.

• Fault Localization Sensitiveness metric assesses the impact of the used fault localization on the repair performance of the APR system.

• Patch Generation Efficiency metrics aim to measure how efficiently APR systems produce plausible or correct patches.

• Bug Diversity metrics exploit intrinsic attributes of bugs to evaluate APR system performance.

• Benchmark Overfitting metrics aim to highlight the difference of APR systems performance between lab settings and real-world scenarios.

In code refactoring, one of the followed approaches is to minimize the number of code smells in addition to a test suite that passes, or both. The term refactoring as introduced by Opdyke [181], means behavior-preserving program transformations for code quality improvement. Soares et al. [182] propose to check refactoring safety by checking errors due to non behavior-preserving transformations. Soares et al. [182] define this type of errors as semantic errors. The current practice to avoid refactoring errors relies on compilation and tests to assure semantics preservation. Soares et al. [182]'s approach starts by identifying common methods between source and target source code (before and after refactoring). It then generates tests on the common methods, run them on source then the target if no test fail, developer will have more confidence the correctness of the refactorings. Wahler et al. [183] use static analysis and three software metrics: the number of duplicates and the number of duplicate lines using using PMD [8] tool, and the number of warning using the tool Findbugs [9]. These three metrics are used for objective evaluation. The usage of objective metrics is combined with software engineers' judgment as subjective evaluation to validate refactorings and to improve the maintainability of their case study.

Da Silva et al. [184] leverage generated unit tests to detect semantic conflicts in different merging scenarios. Differential unit tests are generated and run on commit pairs. Their results show that no conflict was wrongly detected assessing the efficiency of the generated unit tests.

Correa et al. [126] exploit regression testing to assess the correctness of OCL specifications' refactoring.

In literature, different approaches exist for code translation from a programming language to another. Roziere et al. [185] leverage generated unit tests to filter out invalid

8. https://pmd.github.io/
9. https://plugins.jetbrains.com/plugin/4597-qaplug–findbugs

translations and reduce the noise in the generated translations to have better candidates.

3.5 LLMs for co-evolution

Since their appearance in 2022 [10], Large Language Models transformed the computer science industry and the industry of the world, as computer science is concretely used everywhere. In this thesis, that started before this revolution, I found that it is important to investigate the path of LLMs and its intersection with the scope of our work. Thousands of scientific papers are produced in many domains to treat thousands of topics. In this section, I present the most related work to code co-evolution.

Early studies on Copilot focus on the exploration of the security of the generated code [186], comparison of the performances of Copilot with mutation-based code generation techniques [187], and the impact on productivity and the usefulness of Copilot for developers [188], [189]. Nguyen et al. [190] performed an early empirical study on the performance and understandability of Copilot generated code on 34 problems from Leetcode. Doderlein et al. [191] extended the study of Nguyen et al. [190] and run an empirical study on the effect of varying temperature and prompts on the generated code with Copilot and Codex. They used a total of 446 questions to solve from Leetcode and HumanEval data sets. Nathalia et al. [192] evaluated the performance and efficiency of ChatGPT compared to beginners and experts software engineers. Yeticstiren et al. [193] compared the code quality generated from Copilot, CodeWhisperer, and ChatGPT, showing an advantage for ChatGPT in generating correct solutions. Guo et al. [194] ran an empirical study on ChatGPT and its capabilities in refining code based on code reviews. Fu et al. [15] also evaluated ChatGPT and its ability to detect, classify, and repair vulnerable code. Kabir et al. [16] evaluated ChatGPT ability to generate code and to maintain it by improving it based on a new feature description to add in the code. White et al. [195] propose a set of prompt patterns for different tasks that can be used in different phases in the life cycle of a software development, for example: API generation prompt pattern, DSL creation prompt pattern, code quality, and refactoring prompt patterns.

Sridhara et al. [196] explore how ChatGPT can be used in developers' assistance with fifteen common software engineering tasks, among which: method name suggestion, log summarization, Python type inference, commit message generation, vulnerability detection, detection, test oracle generation, and merge conflict resolution. Sridhara et al. [196] found

10. https://www.dataversity.net/a-brief-history-of-large-language-models/

that Extract Method refactorings did not match with the developers' refactorings collected from Silva et al. [197], however, when the authors checked the generated refactoring manually, they found that they are syntactically and semantically correct. Besides code generation and code documentation, the paper of Sadik et al. [198] investigates the potential application ChatGPT as an LLM for bug detection, code refactoring, and bad smell detection. Hemberg et al. [199] explain their Evolutionary Algorithm (EA), with evolutionary operators, can use an LLM to evolve code, how the operators are designed to formulate LLM prompts, and to process LLM responses, while code is represented as a sequence of text in code syntax. Its goal is to get the best solution code that fits the best the beforehand mentioned operators as hyper parameters of the evolutionary algorithm.

Zhang et al.[200] aim to detect code smells in copilot-generated python code and to evaluate copilot capacity in fixing these code smells. Results show that Copilot was able to detect 8 types of code smells out of 10 with 87.1% as fixing rate showing the promising capacity of copilot in fixing python code smells.

Moreover, other studies focused on evaluating LLMs in MDE activities. Chen et al. [18] propose a comparative study between GPT-3.5 and GPT-4 in automatically generating domain models. This work shows that GPT-4 has better modeling results. Chaaben et al. [20] showed how using few-shot learning with GTP3 model can be effective in model completion and in other modeling activities. Camara et al. [19] further assessed how good ChatGPT is in generated UML models. Finally, Abukhalaf [201] run an empirical study on the quality of generated OCL constraints with Codex.

However, these studies focused on the ability of LLMs to generate MDE artifacts, such as models and constraints, but not on their co-evolution. Only Fu et al. [15] looked at repairing vulnerable code with ChatGPT. Jiang et al. [202] proposed self-augmented code generation framework based on LLMs called SelfEvolve. SelfEvolve allows generating code and keep correcting it iteratively with the LLM. finally, Zhang et al. [17] proposed Codeditor, an LLM based tool for code co-evolution between different programming languages. It learns code evolutions as edit sequences and then uses LLMs for multilingual translation.

3.6 Summary and Discussion

After having reviewed the current landscape of metamodel and code co-evolution, this section synthesizes the key findings. It focuses on distilling insights from 1) the co-evolution

Table 3.2 – Related work comparison

Approaches	Category	Approach	Automation	Requires pre-learning	Change types	Validation
Lamothe et al. [147]			Semi-automatic	Yes ✓	Encapsulate, Move method,Remove parameter, Rename, Consolidate, expose implementation, add contextual data,change type, Replaced by exeternal API	No ×
Fazzini et al. [146]	Android api migration	Identifying migration patterns and rank them to select the most context-similar	Fully-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete)	Yes ✓
Meng et al. [155]	Bug fix context		Semi-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete)	No ×
Wu et al. [203]	Java library evolution	Hybrid approach using call dependency graph and textual similarity	Semi-automatic	No ×	change rules : One-to-One, One-to-Many Many-to-One, Simple-Deleted	No ×
Dagenais et al. [129], [130]		Recommendation approach for compilation errors' correction	Semi-automatic	Yes ✓	Deleted or deprecated methods	No ×
Henkel et al. [127]		Catch refactoring operations during the API revolution the API user can replay these operations later	Semi-automatic	Yes ✓	Refactoring operations: Rename Type, Moving Java Elements, Move static member, Change Method Signature, Rename non-virtual method, Rename non-virtual method, Rename virtual method, change type, rename field, Use super-type where possible, Introduce factory	No ×
N. Guyen et al. [131]		Recommendation approach for API usage adaptation in client	Semi-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete)	No ×
Zhong et al. [149]		Compiler-directed tool for migrating API callsite of client code	Fully-automatic	No ✓	N/a	No ×
Gerasimou et al. [133]	Other library Evolution	Code-based transformation to update client code	Semi-automatic	No ×	A set of mapping rule	No ×
Xu et al. [148]		Mining stored database edits to select applicable edits to be reviewed	Fully-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete) classified into 3 categories : Single statement, Block of statements, MultiBlock of statements	No ×
Khelladi et al. [11]	Model-centric evolution	Impact propagation approach	Semi-automatic	No ×	See Table 4.1	No ×
Our approach		Code co-evolution guided by Metamodel changes pattern matching	Fully-automatic	No ×	See Table 4.1	Yes ✓

of metamodels with models, constraints, and transformations, and code, 2) in addition to checking behavioral correctness of code evolution approaches, alongside 3) leveraging LLMs in code co-evolution. This synthesis also highlights the link to the challenges that I mentioned in the introduction 1.

The factors of discussion when talking about the co-evolution of metamodels with models, constraints, and transformations, and code, are: 1) the degree of automation, 2) the impact analysis processing, 3) assessing the behavioral correctness of the co-evolution.

Regarding metamodel and model co-evolution approaches, I remark that most automatic approaches are found under the category of predefined resolution strategies. The other ones are either transformation languages or learning approaches. Even though most approaches use the metamodel changes in the process of adapting models, they do not explicitly study the impact of the metamodel evolution on the models except Hebig et al. [54].

Similarly, for metamodel and constraints approaches. Automatic approached use either predefined operations, pre-selected refactoring operations, or a machine learning approach.

Most of the reviewed approaches of metamodel and transformations do not explicitly study the impact of the metamodel evolution on the transformations, except Kusel, et al. [119].

While evolving metamodels implies structural and possible semantic impact on different artifacts, the main focus is given to structural correctness. After browsing a large amount of papers, I find that only Kusel et al. [125] used Pamomo [117] for semantic correctness verification when evolving OCL expressions.

Regarding the related work addressed in Section 3.3 about code co-evolution, I selected the main related work and established a detailed comparison with it in Table 3.2. My contribution in Chapter 4 distinguishes from these approaches by considering and reasoning on the changes at the metamodel level to match the different pattern usages of the generated code elements (details in Section 4.2.5). This is possible thanks to the abstraction offered by the metamodels. I compare them with the following criteria:

1. Automation: it indicates whether the approach is automatic, semi-automatic, or manual.
2. "Requiring pre-learning": this feature indicates if a given approach is standalone by immediately co-evolving the code or needs previous external code analysis to learn how to co-evolve client code by synthesizing the co-evolution pattern.
3. Changes types: it conveys the changes handled by each approach.
4. Validation: to ensure that the co-evolution did not impact the behavior of the code, a post validation step can be added. This feature indicates if the approach uses any mean of checking behavioral correctness of the code after the co-evolution.

I observe that only three existing approaches are fully automatic [146], [148], [149] and all the rest are semi-automatic [11], [127], [129]–[131], [133], [147], [155], [203]. Only four approaches are standalone without requiring a pre-learning phase before the co-evolution. My approach (Chapter 4) is fully automatic and standalone. Moreover, several different set of changes are handled by each approach, varying from low AST changes to high level composed (refactoring likes) changes as in Table 4.1 in my work.

Only Fazzini et al. [146] proposed to validate the co-evolved android applications with a similar methodology as in my work based on tests' execution Chapter 5).

Regarding different approaches that I browse about code evolution, I find that just Fazzini et al. [146] dedicated a space to check behavioral correctness of the code in Android API migration category. Particularly, I noticed a considerable gap in assessing

the behavioral correctness of metamodel and code co-evolution.

Finally, studies focused on either evaluating the ability of LLMs to generate qualitative code, refining it, repairing it if vulnerable, or augmenting it. However, none of them specifically explored the task of code co-evolution. Other studies also focused on the ability of LLMs to generate MDE artifacts, such as models and constraints, but not on their co-evolution. However, no study investigated the ability of LLMs in the MDE problem of code co-evolution when metamodels evolve. I empirically evaluated how effective is Chagpt in solving this co-evolution problem in Chapter 6.

3.7 Conclusion

To conclude, this chapter has provided a comprehensive overview of existing research in the field of code co-evolution. I started with an exploration of metamodel change detection approaches. I included this section because I find that it is essential to understand the output of these approaches that is used later in the impact analysis of this output on the code. This exploration allowed me to select the approach of Khelladi et al. [60]. Following this, the chapter examined the co-evolution of models, transformations, and constraints in response to evolving metamodels.

The discussion then extended to code co-evolution, covering relevant literature on API-client evolution and language evolution. The chapter also reviewed studies on checking the behavioral correctness of code co-evolution, addressing the challenges of ensuring that modifications do not introduce unintended side effects. Additionally, an overview of the application of Large Language Models (LLMs) in MDE and SLE tasks was presented, showcasing their potential in automating and improving the co-evolution process.

Finally, this chapter concluded with a discussion on the limitations and research gaps identified in the existing literature. This discussion allowed me to study the applicability of these different approaches on the code source artifact, and to better situate our contributions in the context of the existing work when narrowing down the research gaps. These gaps serve as a foundation for the subsequent chapters, guiding the direction of this dissertation toward addressing the challenges in code co-evolution and exploring novel solutions: 1) to enhance metamodel and code co-evolution, 2) to check behavioral correctness of the co-evolution, and 3) to leverage LLMs for code co-evolution in Model-Driven Engineering.

Chapter 4

Automated co-evolution of metamodels and code

One of the key findings of the state of the art chapter 3, is the gap that exists in tackling the problem of metamodel and code co-evolution. Most of the existing literature about evolution in Model-Driven Engineering, concerns the artifacts of models, constraints, and transformations, but rarely code. In this chapter, I present my contribution to address the challenge (C1) about resolving the impact of the metamodel evolution on the code automatically. This chapter is structured as follows. Section 4.1 introduces a motivating example showing the breaking impact of the metamodel evolution on the code. In section 4.2, I present the overall automatic co-evolution approach. This approach starts by using the output changes of the metamodel, then it retrieves the errors that are caused by these changes. Thanks to the pattern matching, for each error a resolution is selected and executed. I first give a brief reminder about metamodel changes' detection and why we selected the approach of Khelladi et al. [68], and how its output is used in the approach. After that, I explain the error retrieval process, followed by a short subsection about the resolution catalog that I use. In the next subsections, I detail the pattern matching phase and the repair mechanism. The overall approach ends with a presentation of the prototype implementation. Section 4.3 details our evaluation that includes four research questions. These research questions aim to 1) assess the applicability of the approach, 2) to measure its efficiency and its behavioral correctness 3) to compare it with IDE quick fixes as a first baseline, and 4) to compare it with the approach of Khelladi et al. [11] as a second baseline. This section ends with the evaluation results, followed by threats to validity in Section 4.4. Finally, Section 4.5 concludes the chapter with a summary and highlights the approach main contributions.

4.1 Motivating Example

This section introduces a motivating example to illustrate the challenge of metamodel and code co-evolution. Let us take as an example the Modisco project [204], which has evolved numerous times in the past. Modisco is an academic initiative project implemented in the Eclipse platform to support the development of model-driven tools, reverse engineering, verification, and transformation of existing software systems [205], [206].

Figure 4.1 shows an excerpt of the "Modisco Discovery Benchmark" metamodel [1] consisting of 10 classes in version 0.9.0. It illustrates some of the domain concepts **Discovery**, **Project**, and **ProjectDiscovery** used for the discovery and reverse engineering of an existing software system. From these metaclasses, a first code API is generated, containing Java interfaces and their implementation classes, a factory, a package, etc. Listing 6.1 shows a snippet of the generated Java interfaces and classes from the metamodel in Figure 4.1.

The generated code API is further enriched by the developers with additional code functionalities in the "Modisco Discovery Benchmark" project and its dependent projects as well. For instance, by implementing the methods defined in metaclasses and advanced functionalities in new classes. Listing 4.2 shows the two classes `Report` and `CDOProjectDiscoveryImpl` of the additional code in the same project "Modisco Dsicovery Benchmark" and in another dependent project, namely the "Modisco Java Discoverer Benchmark" project. In version 0.11.0, the "Modisco Discovery Benchmark" metamodel evolved with several significant changes, among which the following impacting changes:

1. Deleting the metaclass `ProjectDiscovery`.
2. Renaming the property *totalExecutionTimeInSeconds* to *discoveryTimeInSeconds* in metaclass `Discovery`.
3. Moving the property *discoveryTimeInSeconds* (after its rename) from metaclass `Discovery` to `DiscoveryIteration`.

After applying these metamodel changes as illustrated in Figure 4.2, naturally, the code of Listing 4.1 is regenerated from the evolved version of the metamodel, which in turn impacts the existing additional code depicted in Listings 4.2. The resulting errors in the original code in version 0.9.0 are underlined in red in Listing 4.2. Listing 4.3 presents the final result of the co-evolution process in version 0.11.0. The co-evolved code is underlined in green. For example, in response to the *delete* of the metaclass `ProjectDiscovery`, its

1. `https://git.eclipse.org/r/plugins/gitiles/modisco/org.eclipse.modisco/+/refs/tags/0.12.1/org.eclipse.modisco.infra.discovery.benchmark/model/benchmark.ecore`

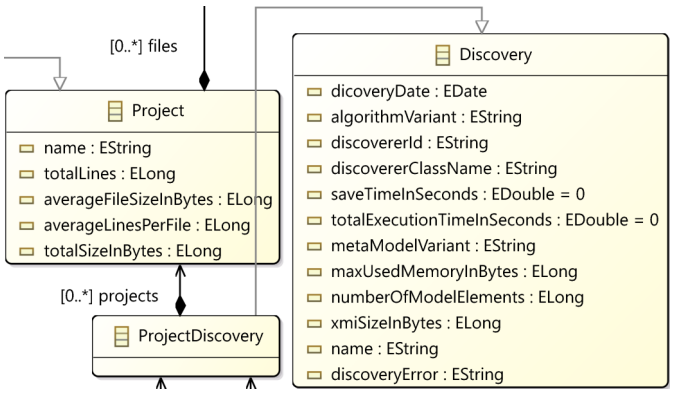


Figure 4.1 – Excerpt of Modisco Benchmark metamodel in version 0.9.0.

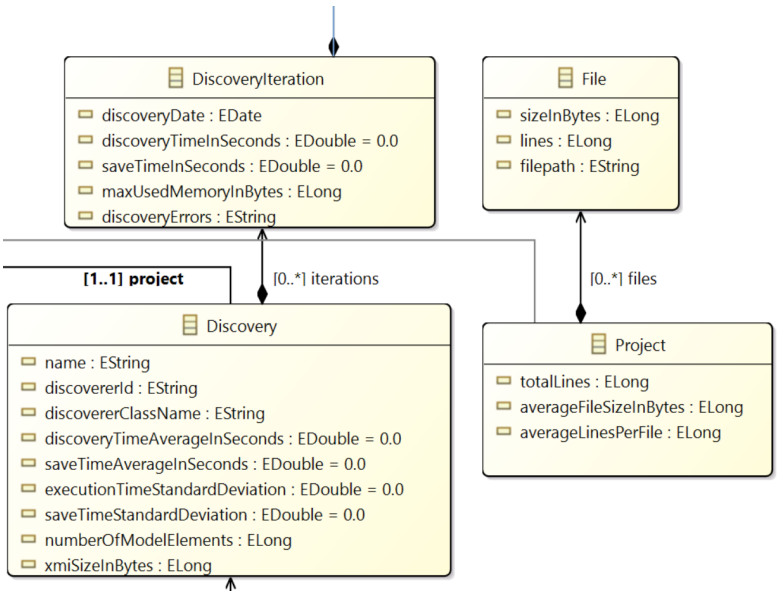


Figure 4.2 – Excerpt of Modisco Benchmark metamodel in version 0.11.0.

import statement *Line* 1 in Listing 4.2, and any usage of it or its methods are impacted. The import statement is completely removed. The same can be applied to the usages of the class and its methods. Alternatively, they could also be replaced by a default value rather than removing the whole instruction. The intention is to maintain the developers' code with minimal removal co-evolution.

Furthermore, the same changes *rename* and *move* of the property *totalExecutionTimeInSeconds* impact two usages that are co-evolved differently. First, the call of `setTotalExecutionTimeInSeconds` (*Line* 4 in Listing 4.2) that is co-evolved by renaming it to `setDiscoveryTimeInSeconds`, then extending the path with `getIterations()`. The second impact is the use of the generated literal `BenchmarkPackage.`*DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS*. It is succes-

sively co-evolved by renaming it to `BenchmarkPackage.`*DISCOVERY__DISCOVERY_TIME_IN_SECONDS* before replacing its source class `DISCOVERY` by `DISCOVERY_ITERATION`. Note that when using the IDE quick fixes to co-evolve these errors, it suggests to create the method `setTotalExecutionTimeInSeconds` in the class `Discovery` and the literal *DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS* in the class `BenchmarkPackage`, which does not meet the required co-evolutions shown in Listing 4.3.

The above examples show the importance of correctly matching the different code usages and patterns of the generated code elements with the metamodel evolution changes to co-evolve them with the appropriate resolutions.

The next section presents our contribution for a fully automatic co-evolution of metamodel and code.

Listing 4.1 – Excerpt of the generated code in org.eclipse.modisco.infra.discovery.benchmark.

```
1  //Discovery Interface
2  public interface Discovery extends EObject {
3      double getTotalExecutionTimeInSeconds();
4      void setTotalExecutionTimeInSeconds(double value);
5      ...
6  }
7  //Project Interface
8  public interface ProjectDiscovery extends Discovery {...}
9  //DiscoveryImpl Class
10 public class DiscoveryImpl extends EObjectImpl implements Discovery {
11     public double getTotalExecutionTimeInSeconds() {...}
12     public void setTotalExecutionTimeInSeconds(double totalExecTime) {...}
13     ...
14 }
```

Listing 4.2 – Excerpt of the additional code V1.

```
1  import (*{\scriptsize org.eclipse.modisco.infra.discovery.benchmark}*).(*\ul{\
       scriptsize ProjectDiscovery}*);
2  public class Report {
3      ...
4      discovery.(*\ul{setTotalExecutionTimeInSeconds}*)(...);
5  }
6  ...
7  public class CDOProjectDiscoveryImpl extends AbstractCDODiscoveryImpl implements
       CDOProjectDiscovery {
8      ...
9      case JavaBenchmarkPackage.
10     CDO_PROJECT_DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS: return BenchmarkPackage.
11     (*\ul{DISCOVERY\_\_TOTAL\_EXECUTION\_TIME\_IN\_SECONDS}*);
12     ...
13 }
```

Listing 4.3 – Excerpt of the additional code V2.

```
1  (*{\st{import }}*)(*{\scriptsize \st{ org.eclipse.modisco.infra.discovery.benchmark.
       ProjectDiscovery}}*);
2  public class Report {
3      ...
4      discovery.(*\ul{getIterations().}*)
5      (*\ul{setDiscoveryTimeInSeconds}*)(...);
6      ...
7  }
8  public class CDOProjectDiscoveryImpl extends AbstractCDODiscoveryImpl implements
       CDOProjectDiscovery {
9      ...
10     case JavaBenchmarkPackage.
11     CDO_PROJECT_DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS: return BenchmarkPackage.
12     (*\ul{DISCOVERY\_ITERATION\_\_DISCOVERY\_TIME\_IN\_SECONDS}*);
13     ...
14 }
15 ...
16}
```

4.2 Approach

This section presents the overall approach of our automated co-evolution of code with evolving metamodels, instantiating on the Ecore technological space. First, we give an overview of the approach and specify the metamodel evolution changes we consider. Then, we present how we retrieve the resulting errors due to metamodel evolution, followed by the regeneration of the code API. After that, we present the pattern matching process, which is an important part of our fully automatic co-evolution approach, before discussing the resolutions of the code errors.

4.2.1 Overview

Figure 4.3 depicts the overall steps for the automatic co-evolution of the metamodel and code, with horizontally separated parts defining chronological order from the top to the bottom. After the generation step (the upper part of Figure 4.3), the evolution of the Ecore metamodel will cause errors in the additional Java code that depends on the API of the newly generated code (the middle part of Figure 4.3). We take as input the evolution changes of the metamodel between the two versions of this metamodel 1. Then, we parse the additional code 2 to retrieve the list of errors. After that, we get to the bottom part of Figure 4.3, both the list of metamodel changes and the list of errors are used as inputs

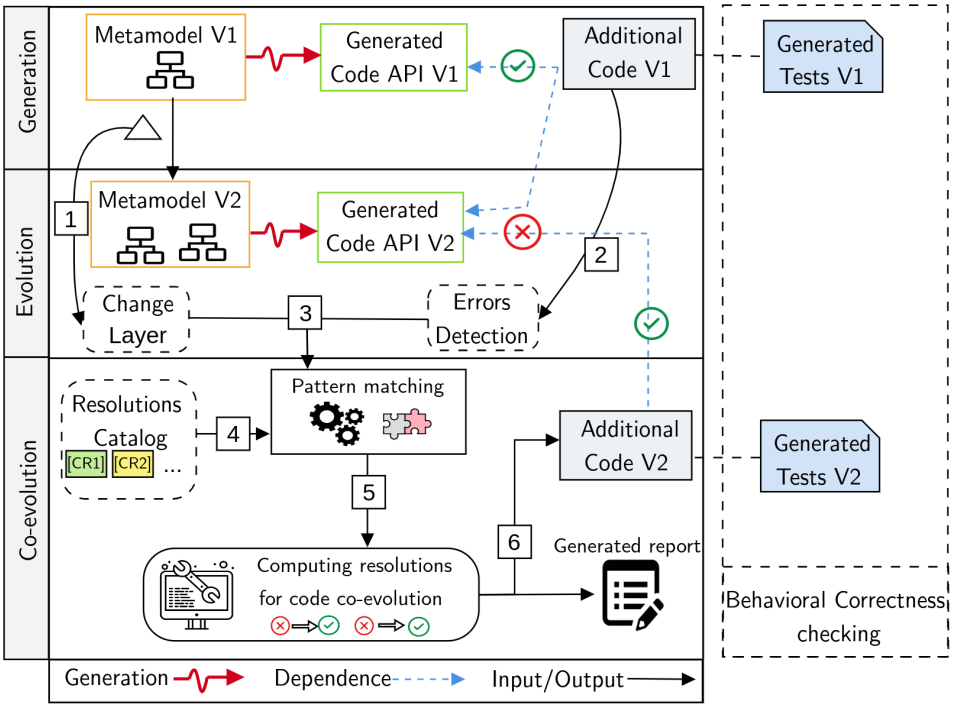


Figure 4.3 – Overall approach for metamodel and code co-evolution

for the pattern matching step 3. It analyzes the structure of the error to match it with its impacting metamodel change and decides which resolution 4 to apply for the error co-evolution 5. The metamodel changes provide the ingredients and necessary information that are used for the co-evolution. At the end of the automatic co-evolution, we obtain a new co-evolved additional code 6 along a generated report on the applied resolutions. In addition to the automatic co-evolution, we generate test cases before and after co-evolution to highlight its possible effect. In fact, many research papers rely on the use of tests to check the behavior of the code during its evolution. For example, Godefroid et al. [207] uses tests to find regressions in different versions of REST APIs. In particular, Lamothe et al. [147], [146] use tests to validate the evolution of the client code after Android API migration. We apply a similar method to check the effect of the co-evolution. Finally, during the co-evolution process, we generate a report linking the applied resolutions for each code error with its impacting metamodel change. If needed, this can help developers in understanding the performed co-evolution, since we fully automate it.

4.2.2 Metamodel Evolution Changes

One of the intrinsic properties of software artifacts is its continuous evolution [53]. Metamodels are no different and are meant to evolve. Two types of evolution changes are considered when evolving a metamodel: *atomic* and *complex* changes [58]. Atomic changes are additions, removals, and updates of a metamodel element. Complex changes consist of a sequence of atomic changes combined together [59], [60]. For example, move property is a complex change where a property is moved from a source class to a target class. This is composed of two atomic changes: delete property and add property [58]. Many approaches in the literature [59], [61]–[66] exist to detect metamodel changes between two versions. Note that the detected list of complex changes does not include the list of the detected atomic changes, i.e., no overlap in between. Moreover, the detection approaches must order the changes in a consistent way. This is fundamentally a problem that change detection approaches must deal with, and hence, is out of scope for our problem of code co-evolution. Nonetheless, it is important and expect a consistent order of changes to not hinder the quality of the co-evolution.

For the purpose of modularity and extensibility, we use a specification layer for the changes 1 that is simply a connection layer to our co-evolution approach with existing change detection approaches. This connection layer specifies our own representation of a metamodel change that can be mapped later with any change representation. It simply specifies the needed information for each change in the form of its attributes. In the left column of Table 4.1, we precise the impacting changes that we consider in our work. For each change, we precise in the second column the attributes that represent and compose each change. When using a state-of-the Art detection approach, we analyze in white box the detected changes to extract their attributes and map them to our internal change layer. For example, a rename property change includes information regarding its old name, new name, and its class container. Therefore, in practice, any detection approach [59], [61]–[66] can be integrated by bridging its changes' representation to our change layer and the rest of co-evolution can be performed independently. In this chapter, we chose to reuse our previous work [65], [208], a heuristic-based approach to actually detect atomic and complex changes between two versions of a metamodel. In the rest of the chapter, we focus on the code co-evolution since it is our main contribution.

4.2.3 Error Retrieval

After the metamodel is evolved and the code API is re-generated, errors will appear in the additional code that must be co-evolved. Unlike code migration context [147], [127], these errors represent the delimited impact of the metamodel evolution. Thus, rather than an impact analysis on the original version to trace the impact of a metamodel change in the code, our approach relies on the compilation result of the code to retrieve its errors. This is necessary and useful in our approach, as we will need to keep updating the list of code errors after co-evolving each given error, hence, iteratively co-evolving the code. We detail this process in the following subsections.

To retrieve those errors, we start by parsing the code of each Java class, called a *compilation unit*, to access the Abstract Syntax Trees (ASTs). An error in a Java code is called a *Marker* that contains the information regarding the detected error. It contains the necessary information to locate the exact impacted AST node in the parsed global AST (*i.e.,* char start and end) and to process it (*i.e.,* message). In the remaining part of the chapter, instead of Markers, compilation units, and additional code, we respectively refer only to errors, Java classes, and code for the sake of simplicity.

4.2.4 Resolution Catalog

Now that we have a list of code errors, we need a set of resolutions to co-evolve them. Our co-evolution approach relies on the resolutions shown in Table 4.1. It depicts the resolutions associated with metamodel changes that are known to have an impact on code [209]. The resolutions are taken from existing co-evolution approaches of various MDE artifacts [39], [71], [92], [103], [106]–[108], [114], [118]–[120], [123], [125], [126], [210]–[213], where they showed to be efficient and useful in co-evolving code [11].

For example, resolutions $[CR8, CR9, CR10, CR11]$ aim to co-evolve the different code errors of a move property in the metamodel.

4.2.5 Pattern Matching for Resolution Selection

As shown in Table 4.1, alternative resolutions exist per metamodel change. Since we aim to fully automate the co-evolution, we need a mechanism to analyze the code error, the code usage, and its impacting metamodel change to decide which resolution to apply. This section presents the pattern matching process between metamodel changes and code usages for the retrieved code errors to automatically select resolutions for their co-evolution.

Table 4.1 – Catalog of resolutions used for the code co-evolution of direct errors due to the metamodel changes.

Impacting Metamodel Changes	Changes' attributes	Proposed Code Resolutions
◇ Delete property p from class C	○ Property p name ○ Container class C name	▷[$CR1$] Remove the direct use of p (e.g., label = s.name + s.m1().p.m2() → label = s.name + ((Type_Of_P) s.m1()).m2()) ▷[$CR2$] Remove the statement using p (i.e., if, loop, assignment, etc.) ▷[$CR3$] Remove the whole call path of p (e.g., label = s.name + s.m1().m2().p → label = s.name) ▷[$CR4$] Replace the whole call path of p with a default value (e.g., id = s.id + s.m1().m2().p → id = s.id + 0)
◇ Delete class `C`	○ Class C name ○ Container package Q name	▷[$CR1$] Remove the direct use of the type `c` (e.g., extending/implementing `c`, in method argument/returned type and not the whole method declaration. Calls to the updated methods are subsequently updated) ▷[$CR2$] Remove the statements using the type `C` (e.g., import, variable declaration, method argument/returned type, method declaration, type instantiation, etc. Calls to the deleted variables and methods are subsequently removed)
◇ Rename element e to e'	○ Element e old name ○ Element e' new name ○ Container package Q or class C name	▷[$CR5$] Rename e in the code
◇ Generalize multiplicity of property p of the class C from a single to multiple values	○ Property p name ○ Container Class C ○ Old multiplicity ○ New multiplicity	▷[$CR6$] Retrieve the first value of a collection (e.g., value = *lng.p* → value = *lng.p.toArray()[0]* or *lng.p.get(0)*)
◇ Move property p_i from class `S` to `T` through *ref* ◇ Extract class of properties p_1, ..., p_n from `S` to `T` through *ref*	○ Property p name ○ Source container class S name ○ Target container class T name ○ Reference *ref* name	▷[$CR7$] Extend navigation path of p_i (e.g., $lng.p_i \rightarrow lng.ref.p_i$) ▷[$CR8$] Extend navigation path of p_i and add a for loop (e.g., $lng.p_i \rightarrow for(v\ in\ lng.ref)\ \{v.p_i\}$) ▷[$CR9$] Reduce navigation path of p_i (e.g., $lng.ref.p_i \rightarrow lng.p_i$) ▷[$CR10$] Replace S by T_REF in Literal values (e.g., $MetamodelPackage.S\_\_\_p_i \rightarrow MetamodelPackage.T\_\_\_p_i$)
◇ Push property p from class `Sup` to $\texttt{Sub}_1$,...,$\texttt{Sub}_n$	○ Property p name ○ Container class *Sup* name ○ List of container classes Sub_i names	▷[$CR11$] Introduce a type test with an If statement (e.g., $t.name = s.p.name \rightarrow if(s.p.istypeof(Sub_1))$ {t.name = (Sub_1 s).p.name} ... *else* $if(s.p.istypeof(Sub_n)$ {t.name = (Sub_n s).p.name}) ▷[$CR12$] Cast p to one specific sub class Sub_i (e.g., $t.name = s.p.name \rightarrow t.name = ((Sub_i)s).p.name$) ▷[$CR13$] Duplicate the statement using the literal for each subclass and replace Sup by Sub_i (e.g., $add(Package.Sup\_\_\_P) \rightarrow add(Package.Sub_0\_\_\_P)$, ... , $add(Package.Sub_n\_\_\_P)$)
◇ Pull property p from classes $\texttt{Sub}_1$,...,$\texttt{Sub}_n$ to `Sup`	○ Property p name ○ List of container classes Sub_i ○ Container class *Sup* name	▷[$CR14$] Replace Sub_i by Sup in Literal values (e.g., $MetamodelPackage.Sub_i\_\_\_P \rightarrow MetamodelPackage.Sup\_\_\_P$)
◇ Inline class `S` to `T` with properties $p_1, ..., p_n$	○ List of properties p_i ○ Container Source class S name ○ Container Target class T name	▷[$CR9$] Reduce navigation path of p_i (e.g., $lng.ref.p_i \rightarrow lng.p_i$) ▷[$CR15$] Change the class type from `S` to `T` (e.g., List<S> l = ...; → List<T> l = ...;)
◇ Change property p type of the class C from `S` to `T`	○ Property p name ○ Container class C name ○ Old type S ○ New type T	▷[$CR16$] Change variable declaration type initialized with p from `S` to `T` (e.g., S var = s.p; → T var = s.p;) ▷[$CR17$] Add a cast of p

Each Metamodel Element ME has corresponding Generated Code Elements {GCE_0, GCE_1, ..., GCE_n} in the code API. GCE_i have **usages** in the **additional code** as illustrated in Figure 4.4. Thus, evolving a ME will regenerate {GCE_0, GCE_1, ..., GCE_n} which will impact their **usages**.

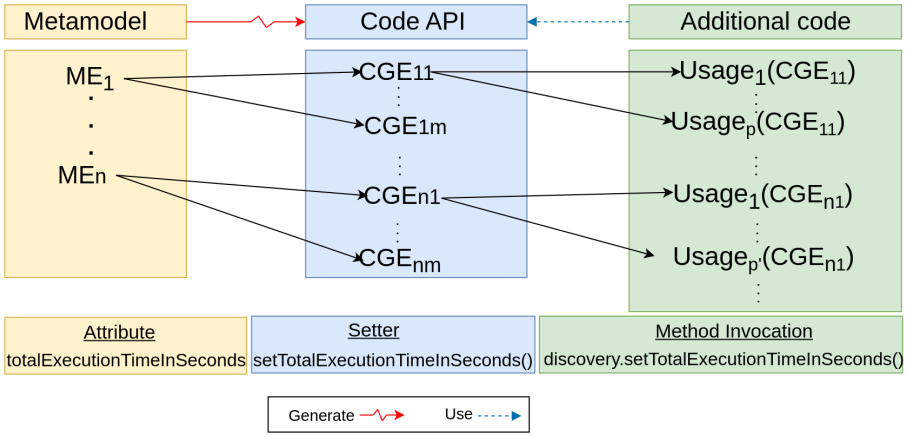


Figure 4.4 – Schema for mapping between the metamodel and code.

Table 4.2 classifies the different patterns of the generated code elements GCE_i for each metamodel element ME type, and provides illustrative examples. It shows that various patterns of code elements are generated for each metamodel element type. For example, let us consider the case of a metaclass. EMF generates a corresponding interface and a class implementation, a *createClass()* method in the factory class, three literals (*i.e.,* constants) for the class and an accessor method in the package class, and a corresponding create adapter method. For the attribute case, EMF generates the signature and implementation of a getter and a setter, an accessor method in the package class, and a literal.

This classification is essential to match the different code errors with their corresponding **pattern** of the GCE_i in Table 4.2. Moreover, each generated code element GCE_i can be used in different **configurations** in the code that must also be considered in the pattern matching process. For example, using a GCE_i as a parameter in a method declaration and in a method invocation, or to initialize a variable declaration, or in an expression call in a statement, etc., are considered as different configurations and can influence which resolution to apply as well. With these ingredients, we can match a resolution for each error.

Algorithm 1 summarizes the pattern matching process. Given a Java class, an error, and a list of metamodel changes, Algorithm 1 first retrieves the error AST node *line* 2. After that, it identifies the configuration of the GCE usages *Line* 3. Then, for each

Table 4.2 – Classification of the different patterns of the generated code element from the metamodel elements.

Metamodel element type	**Generated code elements**	**Pattern of the generated code elements**	**Illustrative examples**
Metaclass	Interface	"MetaClassName"	*Constraint*
	createClass() (in metamodelFactory class)	"create"+"MetaClassName"()	*createConstraint()*
	Literals of the class	"META_CLASS_NAME" "META_CLASS_NAME"+"_"+ "FEATURE_COUNT" "META_CLASS_NAME"+ "_"+"OPERATION_COUNT"	*CONSTRAINT*, *CONSTRAINT_FEATURE_COUNT*, *CONSTRAINT_OPERATION_COUNT*
	Accessor of Meta objects (in metamodelPackage class)	"get"+"MetaClassName"()	*getConstraint()*
	Class implementation	"MetaClassNameImpl"	*ConstraintImpl*
	Adapter	"create"+"MetaClassName"+"Adapter"	*createConstraintAdapter()*
Attribute (same for a reference)	Signature of getters and setters	"get"+"AttributeName"(), "set"+"AttributeName"()	*getStereotype()*, *setStereotype()*
	Accessor of Meta objects	"get"+"MetaClassName"+"_"+"AttributeName"()	*getConstraint_Stereotype()*
	Literal	"META_CLASS_NAME"+"___"+"ATTRIBUTE_NAME"	*CONSTRAINT___STEREOTYPE*
	Implementation of getters and setters	"get"+"AttributeName"(), "set"+"AttributeName"()	getStereotype(), setStereotype()
Method	Declaration of the method	"methodName"()	*UniqueName()*
	Accessor of meta objects	"get"+"MetaClass"+"___"+"MethodName"()	*getCONSTRAINT___UniqueName()*
	Literal	"META_CLASS_NAME"+"___"+"METHOD_NAME"	*CONSTRAINT____UNIQUE_NAME*
	Implementation of the method	"methodName"()	*UniqueName()*

metamodel change type $\boxed{lines\ 6, 17, 22, 31}$, it identifies the pattern of the corresponding GCE presented in Table 6.1 $\boxed{line\ 7, 18, 23, 32}$. Depending on the detected configuration, the appropriate pre-selected resolution is added to the output set $\boxed{Line\ 10, 13, 19, 26, 35}$. Finally, the selected resolutions set is returned for further processing in the automatic co-evolution $\boxed{Line\ 41}$. Let us take the example of `"Change property type from S to T"` (last change type in Table 4.1). It has two possible resolutions, CR16 and CR17. Assuming that we have a code error that must be co-evolved. Our pattern matching process first identifies the pattern of the corresponding GCE_i from Table 6.1 (Line 7), then the configuration of the GCE_i usage (Line 8).

Algorithm 1 starts by parsing the error to find the pattern of the corresponding GCE_i, which will help to find the causing change. The next step is to define the configuration of the GCE_i which means the type of the GCE_i usage. If it is a variable declaration (line 9), the resolution $CR16$ is returned. If any other configuration is detected, the pattern matching process returns $CR17$ as an appropriate resolution.

Note that an error can be matched with more than one resolution because a metamodel element ME can be impacted by more than one change, in other terms interdependent changes. For example, Algorithm 1 allows matching the error in $\boxed{Line17}$ (rename property)

and $\boxed{Line31}$ (move property) with two metamodel changes of the property *totalExecutionTimeInSeconds* in Listing 4.2. The generated literal, which is our generated code element GCE_i, is used here in the configuration of a literal static field access *BenchmarkPackage.DISCOVERY__TOTAL_EXECUTION_TIME_IN_SECONDS*. The returned resolutions are $CR5$ and $CR10$ $\boxed{Line19, 35}$, to be executed in the order of detection of their causing metamodel changes.

A similar mechanism is implemented for the rest of metamodel changes to select a resolution based on the pattern of GCE_i and the configuration of its usage. For the sake of readability, Algorithm 1 does not show all possible combinations of *metamodel changes* $\times$ *patterns* $\times$ $configurations$, but few examples to illustrate its essence. We nonetheless give an extended version in the appendix.

Finally, in our implementation, for the case of a "Delete Class" and "Delete Property", we favor the least deletion when possible. In particular, depending on the configuration usage, we select the resolution that deletes the least possible among $CR1$, $CR2$, $CR3$, $CR4$. For example, for an error in a parameter of a method call, we select the resolution $CR4$ rather than $CR1$ or $CR2$.

In Listing 4.2, Algorithm 1 matches the error in the import declaration (**Configuration**), with the deletion of the metaclass *ProjectDiscovery* (**pattern**), which allows to select the resolution $CR2$.

Algorithm 1: Pattern matching algorithm

```
Data: javaClass, error, changesList
resolution_s ← φ
errorNode ← findErrorAstNode(javaClass, error)
configuration ← getConfiguration(javaClass,errornode)
for (change ∈ changesList) do
    switch change do
        case ChangePropertyType do
            if (match(patternGCE,change) then
                switch configuration do
                    case VariableDeclaration do
                        resolution_s.add("CR16")
                    end
                    otherwise do
                        resolution_s.add("CR17")
                    end
                end
            end
        case RenameProperty do
            if (match(patternGCE,change) then
                resolution_s.add("CR5")
            end
        case DeleteClass do
            if (match(patternGCE,change) then
                switch configuration do
                    case ImportDeclaration do
                        resolution_s.add("CR2")
                    end
                    ... /*Other configurations*/
                end
            end
        case MoveProperty do
            if (match(patternGCE,change) then
                switch configuration do
                    case LiteralStaticField do
                        resolution_s.add("CR10")
                    end
                    ... /*Other configurations*/
                end
            end
        case ... /*Other changes*/ do
            ...
        end
    end
end
return resolution_s
```

4.2.6 Repair mechanism for the code co-evolution

After the pattern matching step, we can proceed to co-evolve the code. Herein, we distinguish **direct errors** and **indirect errors**. The former are errors that do use a Generated Code Element GCE. They can be matched with one or many metamodel changes to select the appropriate one or many resolution(s). The latter are errors that do not use a generated code element. They are not matched, and hence, cannot be resolved using the pattern matching process.

Algorithm 2 presents the general process of the code co-evolution. It starts by parsing the project Java classes $\boxed{line\ 1}$ to be able to access the AST of the code. Then it browses the parsed classes to retrieve the list of errors $\boxed{line\ 3}$. The next step is to run the pattern matching Algorithm 1 for each error to return the matched resolution(s) if any $\boxed{lines\ 4-15}$.

If an error is matched with at least one resolution $\boxed{line\ 7}$, it is a **direct error**. If an error is matched with several changes $\boxed{lines\ 8-10}$, it will be resolved iteratively for each resolution $\boxed{lines\ 9}$.

In the case of an **indirect error**, we cannot apply the pattern matching process, but we still attempt to repair them with the available quick fixes in the IDE. We analyze the error message to match it with one of the proposed Java quick fixes $\boxed{lines\ 11}$. For example, *the type MC must implement the inherited abstract method* is considered as an indirect error. This error can occur when a method is added in a metaclass `MC`. Classes that implement the generated interface generated from `MC` must override the new method. We attempt to repair it with its quick fix *add the unimplemented method* $\boxed{lines\ 12}$.

After applying the resolutions or a quick fix, the Java class has to be refreshed. This is because the modifications of the AST will impact the list of errors and their locations in the Java class $\boxed{lines\ 13-14}$. When refreshing the Java class, the list of errors typically decreases as they are co-evolved, yet, new ones may be introduced. Consequently, they will be handled in the next iterations similarly with our pattern matching and co-evolution algorithm or with the quick fixes.

Finally, this process is repeated until all errors are handled. However, we implemented a stopping criteria to handle the indirect errors with quick fixes. It stops in two cases: 1) when only errors without any quick fix proposal remain, or 2) when the application of a quick fix causes an infinite loop [214], [215], i.e., a cycle of applying a quick fix that causes a previously fixed error (A $\mapsto$ B $\mapsto$ A $\mapsto$ B $\mapsto$ ...). However, as we will see in the evaluation, these cases never happened in our executions.

Algorithm 2: Co-evolution of metamodel and code

```
   Data: EcoreModelingProject, changesList
 1 javaClasses ← Parse(EcoreModelingProject)
 2 for ( jc ∈ javaClasses) do
 3     errorsList ← getErrors(jc)
 4     while (!errorsList.isEmpty()) do
 5         error ← errorsList.next()
 6         resolution_s ← patternMatching(jc,error, changesList)
 7         if (! resolution_s.isEmpty()) /*direct errors*/ then
 8             for (resolution ∈ resolution_s) do
 9                 applyResolution(jc,error, resolution)
10             end
11         else if error.hasQuickFix() /*indirect errors*/ then
12             useQuickFixes(error)
13         refreshJavaClass(jc)
14         refreshErrorsList(jc, errorsList)
15     end
16 end
```

Note that during the co-evolution process, we log the history of execution: the Java class, the error, its line, the change that provoked it, and the applied resolution(s)/quick fix in a generated report. Thus, we can generate a detailed report that allows the developer to check the modifications applied to the co-evolved code after the co-evolution is finished, i.e, their validation is not mendatory to concretly apply the resolutions. Table 4.3 shows an example of a report for the corresponding part to our motivation example from Modisco Java Discoverer Benchmark applied co-evolution.

Table 4.3 – Excerpt from the traced report of Modisco Java Discoverer Benchmark project

File	Error	Line	Change	Resolution
CDOProjectDiscoveryImpl.java	ProjectDiscovery	29	Delete class ProjectDiscovery	CR2
Report.java	setTotalExecutionTimeInSeconds	183	Rename property	CR5
Report.java	setDiscoveryTimeInSeconds	183	Moving property	CR7
CDOProjectDiscoveryImpl.java	DISCOVERY___TOTAL_EXECUTION_TIME_IN_SECONDS	799	Rename property	CR5
CDOProjectDiscoveryImpl.java	DISCOVERY___DISCOVERY_TIME_IN_SECONDS	799	Move property	CR10

4.2.7 Prototype Implementation

We implemented our solution as an Eclipse Java plugin handling Ecore/EMF metamodels and their Java code.

The co-evolution process technically consists of the code AST manipulation using the JDT eclipse plugin [2].

— **Error retrieving**: there are many methods to manipulate the compilation errors of the code. We use the marker of IJavaModelMarker.JAVA_MODEL_PROBLEM_MARKER. Then we filter markers whose severity value equals 2.

— **Change detection layer**: it consists of a set of model classes that specifies the information of each type of atomic and complex changes.

— **Pattern matching**: to match the error AST node with the causing change, we proceed by string formatting between the identifier of the AST node and the relevant information encapsulated in the change. We find the corresponding configuration by looking for the higher levels of the error AST node in the code AST using the package org.eclipse.jdt.core.dom.

— **Resolutions** : The AST nodes of errors are manipulated with edit actions: modification, replacement, or deletion) using the package *org.eclipse.jdt.core.dom.rewrite*, org.eclipse.text.edits.TextEdit, and *org.eclipse.core.filebuffers*.

— **Quick fixes** : we use *org.eclipse.jdt.ui.text.java.IQuickAssistProcessor* and *org.eclipse.jdt.ui.text.java.IJavaCompletionProposal*.

4.3 Evaluation

This section evaluates our automatic co-evolution approach. First, we present the evaluation process and the data set. Then, we set the research questions we address and discuss the obtained results.

4.3.1 Evaluation Process

We evaluate our automated co-evolution of code with metamodel evolution by measuring: 1) its ability to co-evolve the errors, 2) its time performance, 3) its correctness

2. Eclipse Java development tools (JDT): `https://www.eclipse.org/jdt/core/`

by using recall and precision metrics, 4) its behavioral correctness by using test suites before/after the automatic co-evolution, 5) and by comparing it with both quick fixes and our prior work [11].

First, as our approach co-evolves the erroneous code due to metamodel evolution, we need to **provoke the errors in the code**. To do so, we replace the original metamodel with the evolved metamodel. Then, we regenerate the code API with EMF. This will cause errors in the code that our approach must co-evolve (1). We then use the function *System.nanoTime*() for time measurement (2). After that, we measure the correctness of our code co-evolution (3) by comparing, for the same set of code errors that we automatically co-evolved, how they were manually co-evolved by developers. This allows us to measure the *precision* and *recall* reached by our co-evolution approach. They vary from 0 to 1, i.e., 0% to 100%. They are defined as follows:

$$precision = \frac{AppliedResolutions \cap ExpectedResolutions}{AppliedResolutions}$$

$$recall = \frac{AppliedResolutions \cap ExpectedResolutions}{ExpectedResolutions}$$

The *AppliedResolutions* are those by our approach (from Table 4.1) and the *ExpectedResolutions* are the actual manually performed resolutions by developers.

Moreover, to check if the co-evolution impacts the original code behavior (4), we generate tests for the original and the automatically co-evolved versions. Hence, we observe the behavioral effect of our co-evolution through the tests. To do so, we rely on EvoSuite [158], a popular tool for test cases generation, as it is widely used by researchers and developers. Gruber et al. [216] further showed the quality and robustness of the generated tests. It showed that while flakiness is at least as common in generated tests as in developer-written tests, EvoSuite is effective in alleviating this issue giving 71.7% fewer flaky tests. Thus, EvoSuite is appropriate in our work to generate robust tests in the original code and in the co-evolved code to compare their results, i.e., check behavioral correctness.

Furthermore, we compare our approach with the application of quick fixes (5). For each error, we apply the first quick fix in the list of corresponding proposals, then we compare the precision and recall of quick fixes with the precision and recall of our approach. Finally, we compare our approach with our prior work [11] in terms of precision and recall, and general rationale (5).

4.3.2 Data Set

This section presents the used data set in our evaluation to be found in the attached supplementary material [3]. We chose the EMF case study from the Eclipse platform, which is a popular tool that provides a flexible and extensible platform for creating custom modeling tools and applications. In 2022, Eclipse IDE was downloaded 1 million times per month.

First, we aimed at selecting meaningful evolutions that do not consist of only deleting metamodel elements, but rather include complex evolution changes. This selection criterion resulted in projects that do not contain unit tests, and this is the main reason behind relying on generated tests to check the behavioral correctness of the co-evolution. Moreover, to make sure that the errors are due only to the evolution of a single metamodel at a time, we selected projects that were dependent on one metamodel and not dependent (directly or transitively) on several metamodels that evolved simultaneously. This gives us more confidence in observing the resulting errors in the code due only to metamodel changes. Thus, mitigating the bias related to the ambiguous cases where errors interact and mask each other [217]. Handling the scenario code co-evolution due to multiple metamodels evolution is left for future work.

Therefore, we evaluated our approach on nine 9 Java projects from three case studies of three different language implementations in Eclipse, namely OCL [218], Modisco [204], and Papyrus [219]. OCL is a standard language defined by the Object Management Group (OMG) to specify First-order logic constraints. Modisco is an academic initiative to support the development of model-driven tools, reverse engineering, verification, and transformation of existing software systems. Papyrus is an industrial project led by CEA [4] to support model-based simulation, formal testing, safety analysis, etc. Thus, the three case studies cover standard, academic, and industrial languages that have evolved several times for more than 10 years of continuous development period. In particular, Papyrus and OCL are open-source projects that are actively maintained with frequent releases per year.

Table 4.4 gives details on the selected case studies, in particular about their metamodels and the changes applied during evolution. The total of applied metamodel changes was 330 atomic changes, including 19 complex changes in the three metamodels. Note that these real-world metamodel evolution changes do not cover all the changes in Table 4.1. However, we did not force any other missing resolutions to be able to compute recall

3. `https://figshare.com/s/8986914e924300be77da`
4. `http://www-list.cea.fr/en/`

Table 4.4 – Details of the metamodels and their evolutions.

Case study	Evolved metamodels	Versions	Atomic changes in the metamodel	Complex changes in the metamodel
OCL	Pivot.ecore in project ocl.examples.pivot	3.2.2 to 3.4.4	Deletes: 2 classes, 16 properties, 6 super types Renames: 1 class, 5 properties Property changes: 4 types; 2 multiplicities Adds: 25 classes, 121 properties, 36 super types	1 pull property 2 push properties
Modisco	Benchmark.ecore in project modisco.infra.discovery.benchmark	0.9.0 to 0.13.0	Deletes: 6 classes, 19 properties, 5 super types Renames: 5 properties Adds: 7 classes, 24 properties, 4 super types	4 moves property 6 pull property 1 extract class 1 extract super class
Papyrus	ExtendedTypes.ecore in project papyrus.infra.extendedtypes	0.9.0 to 1.1.0	Deletes: 10 properties, 2 super types Renames: 3 classes, 2 properties Adds: 8 classes, 9 properties, 8 super types	2 pull property 1 push property 1 extract super class

Table 4.5 – Details of the projects and their caused direct and indirect errors by the metamodels evolution.

Evolved metamodels	Projects to co-evolve in response to the evolved metamodels	N^o of packages	N^o of classes	N^o of LOC	N^o of Impacted classes	N^o of total direct errors	N^o of tota indirect err
OCL Pivot.ecore	[$P1$] ocl.examples.pivot	22	439	74002	56	489	37
	[$P2$] ocl.examples.xtext.base	12	181	17599	10	27	2
Modisco Benchmark.ecore	[$P3$] modisco.infra.discovery.benchmark	3	28	2333	1	6	0
	[$P4$] gmt.modisco.java.discoverer.benchmark	8	21	1947	4	30	0
	[$P5$] modisco.java.discoverer.benchmark	10	28	2794	9	56	0
	[$P6$] modisco.java.discoverer.benchmark.javaBenchmark	3	16	1654	9	58	15
Papyrus ExtendedTypes.ecore	[$P7$] papyrus.infra.extendedtypes	8	37	2057	8	59	0
	[$P8$] papyrus.infra.extendedtypes.emf	7	12	374	7	23	6
	[$P9$] papyrus.uml.tools.extendedtypes	7	15	725	7	23	6

and precision relatively to the real manual co-evolved code, and to minimize the bias in results. Nonetheless, as a sanity check, we did synthetically try all of them during their implementation before the evaluation. We simulated all metamodel changes with all patterns of generated code elements and configuration usages.

For those three case studies, we collect nine Java projects impacted by those three evolving metamodels and their regenerated code API. We collect the **original** and the **developers' evolved** Java code for those projects. Table **??** gives details on the size of the projects and code of the original versions that we co-evolve. In addition, it gives the number of direct and indirect errors after the metamodel evolution. Finally, Table 4.8's first line gives the total number of tests in the original and the evolved versions of the projects.

4.3.3 Research Questions

This section sets the research questions (RQs) to assess our work. The research questions are as follows:

RQ1. *Can our automatic co-evolution approach handle the code errors by resolving them after the metamodel evolution?* This aims to assess the ability and applicability of our automatic approach to co-evolve the code due to evolving metamodels.

RQ2. *To what extent does our automatic approach correctly co-evolve the erroneous code?* This aims to assess the usefulness and measure the precision and recall of our approach when compared to the manually applied co-evolution for direct code errors. It further assesses the behavioral correctness of the co-evolution by observing the tests' execution before and after the co-evolution.

RQ3. *How does our automatic approach for code co-evolution compare to the IDE quick fixes as a baseline?* This aims to assess the ability of the IDE quick fixes to correctly co-evolve the code due to evolving metamodels. We measure and compare the precision and recall of our automatic co-evolution approach to the quick fixes.

RQ4. *How does our automatic approach for code co-evolution compare to the state-of-the-art semi-automatic approach as a baseline?* This aims to assess the ability of the fully automatic co-evolution compared to a semi-automatic co-evolution [11].

4.3.4 Results

We now discuss the results w.r.t. our research questions.

RQ1. Can our automatic co-evolution approach handle the code errors by resolving them after the metamodel evolution?

Following the evaluation protocol and after regenerating the code API from the metamodels, we observed 837 errors, among which 771 (92%) direct and 66 (8%) indirect errors. Regarding the 771 direct errors, a total of 631 resolutions were applied. This shows the applicability of our co-evolution approach which was able to handle 100% of direct errors in the code caused by the metamodel evolution changes. Regarding indirect errors, 5 quick fixes all of *add the unimplemented methods* were applied to repair 100% of the 66 errors. Thus, after automating completely the co-evolution of the code, the developers are always able to consult the generated report and check the history of co-evolution.

Moreover, we observed that the number of applied resolutions is less than the initial number of direct errors in the code. In fact, as code co-evolution advances, some resolutions repair multiple other errors as a side effect [214], [215]. In particular, we observed the case of renaming the type of a declared variable, the resolution $[CR5]$ is applied. As a consequence, all the erroneous usages of this variable were automatically corrected. We also observed the main case of a delete resolution $[CR2]$ of an instruction that contained several errors in its body. For example, if an error occurs in the condition statement of an `IF` and a `For` instructions, the whole `IF`, or a `For` block is deleted. As a result, all errors in their bodies automatically disappear. However, even if we would have co-evolved the inner errors first, we would have ended up by deleting the `IF` and `For` instructions to co-evolve its parent error. Therefore, our automatic co-evolution would have reached the same code state.

Furthermore, Table 4.6 lists the applied resolutions for each co-evolved project. In total, the 631 applied resolutions represented 11 out of the 17 resolutions from our catalog in Table 4.1. In particular, $32 \times [CR1], 240 \times [CR2], 73 \times [CR4], 179 \times [CR5], 16 \times [CR7], 20 \times [CR10], 18 \times [CR11], 7 \times [CR13], 2 \times [CR14], 3 \times [CR16], 13 \times [CR17]$. Figure 4.5 further illustrates the application frequency of each resolution. Some resolutions were never applied, such as $[CR3], [CR6], [CR15]$. This can be explained by two reasons. The first one is that the corresponding change did never occur in our case studies (Table 4.1) like $[CR15]$. The second reason is that pattern matching favors the least deletion when possible. In particular, $[CR1]$ over $[CR3]$, or $[CR4]$ over $[CR2]$.

Finally, the total co-evolution time per project ranged from a few seconds to almost 10 minutes, respectively, in Modisco *[P3]* and OCL Pivot *[P1]*. On average, the co-evolution per error took less than half a second. The evaluation was run on a Fedora Linux 35 laptop with a Core i9 2.6 GHz and 16 GB RAM.

RQ_1 insights:
We can automatically co-evolve all direct errors in the code after metamodel evolution. Indirect errors are repaired with quick fixes. This allows full automation of the co-evolution with possible manual intervention by checking the generated report.

Table 4.6 – Number of applied resolutions in our code co-evolution for each project and per evolved metamodel.

Evolved metamodels	Co-evolved projects	N^o of patterns	N^o of applied resolutions
OCL Pivot.ecore	$[P1]$	381	$[CR1]:12, [CR2]:176, [CR4]:50,$ $[CR5]:110, [CR11]:13, [CR13]:7,$ $[CR14]:2, [CR16]:2, [CR17]:9$
	$[P2]$	25	$[CR1]:1, [CR2]:7, [CR4]:7,$ $[CR5]:5, [CR11]:4, [CR16]:1$
Modisco Benchmark. ecore	$[P3]$	6	$[CR2]:6$
	$[P4]$	22	$[CR1]:5, [CR2]:13,$ $[CR5]:1, [CR7]:3$
	$[P5]$	50	$[CR1]:6, [CR2]:23, [CR5]:6,$ $[CR7]:13, [CR17]:2$
	$[P6]$	62	$[CR1]:8, [CR2]:14, [CR4]:8$ $[CR5]:12, [CR10]:20, [CR17]:2$
Papyrus ExtendedTypes. ecore	$[P7]$	55	$[CR2]:1, [CR4]:8,$ $[CR5]:45, [CR11]:1$
	$[P8]$	15	$[CR5]:15$
	$[P9]$	15	$[CR5]:15$

RQ2. To what extent does our automatic approach correctly co-evolve the erroneous code?

To assess and measure the precision and recall of our automatic co-evolution, we first compared it with the manual co-evolution of the code that developers went through.

Table 4.7 depicts the reached precision and recall of our automatic co-evolution approach for our nine case studies. We observe that the measured precision and recall varied, respectively from 48% to 100%, and from 51% to 100% reaching an average of,

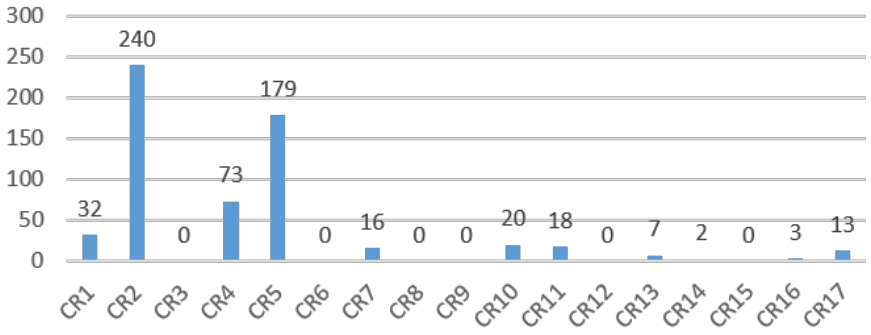


Figure 4.5 – Frequency of applied resolutions overall.

Table 4.7 – Measured precision and recall of our projects.

Projects	P1	P2	P3	P4	P5	P6	P7	P8	P9
precision	90%	80%	100%	81%	58%	48%	98%	93%	93%
recall	66%	83%	100%	75%	58%	51%	94%	100%	100%

respectively 82% precision and 81% recall. This shows the usefulness of the considered resolutions in Table 4.1 for the automatic co-evolution of the direct code errors in our case studies.

We further investigated the cause of lowering precision and recall, i.e., the cases where our automatic co-evolution did not match the expected resolutions. The main observation is that several errors that could have been co-evolved by maintaining them were deleted by the developers in the manually evolved version of the code. Rather than to delete the erroneous code, our approach was able to successfully co-evolve and maintain it. For example, in the project *P5*, we observed errors in the code due to moves of the properties *maxUsedMemoryInBytes* and *totalExecutionTimeInSeconds* and its rename to *discoveryTimeInSeconds*. Our approach automatically co-evolved them and maintained the erroneous code, by renaming the setter and extending its path. Whereas, the developers' manual co-evolution consisted of deleting the impacted code. This may hint at a lack of automated co-evolution support that would have easily maintained the code in the new version rather than deleting it. Thus, our co-evolutions here are valid alternatives even if it did not match the correct manual co-evolution.

It is worth noting that the order of the resolutions' application is following the order of the treated metamodel changes. It may affect the number of iterations of Algorithm 2 but it will neither affect the value of recall and precision nor the final result of the co-evolved code.

Finally, to check the behavioral effect of the co-evolution, we observed the tests' execution that we generated for the original and co-evolved code. Table 4.8 depicts the test execution results. In *[P1, P5, P6, P8, and P9]* similar tests were generated for the original and co-evolved versions. Whereas in *[P2, P3, P7]* there were common tests as well as different generated tests, while we could also not generate tests for *[P4]*. In most cases, we observe that the percentages of passing, failing, and erroneous tests were the same with no significant changes after the co-evolution. Therefore, it suggests that our automatic co-evolution is behaviorally correct by not altering the code behavior of the original projects.

Table 4.8 – Observed test before and after co-evolution. [Legend: Before (V1) – After (V2)]

Projects	P1	P2	P3	P5	P6	P7	P8	P9
N^o tests	1987 – 1987	2261 – 2073	475 – 555	67 – 67	427 – 427	142 – 251	105 – 105	75 – 75
N^o pass	826 – 826 (41% – 41%)	1221 – 1161 (54% – 56%)	349 – 417 (73% – 75%)	32 – 32 (47% – 47%)	2 – 2 (0.4% – 0.4%)	18 – 103 (12% – 41%)	16 – 16 (18% – 17%)	14 – 13 (15% – 15%)
N^o fail	14 – 14 (0.7%–0.7%)	36 – 16 (1,6% – 0,8%)	3 – 3 (0.6% – 0,5%)	2 – 2 (2.9% – 2.9%)	0 – 0 (0% – 0%)	3 – 4 (2.1% – 1.5%)	3 – 3 (2.8% – 2.8%)	1 – 1 (1.3% – 1.3%)
N^o error	1147 – 1147 (57%–57%)	1004 – 896 (44% – 43%)	123 – 135 (25.8% – 24.3%)	33 – 33 (49.2% – 49.2%)	425 – 425 (99.5% – 99.5%)	121 – 144 (85.2% – 57.3%)	86 – 86 (81.9% – 81.9%)	60 – 61 (80% – 81.3%)

***RQ*$_2$ insights:**
Our automatic co-evolution reached an average of an 82% precision and 81% recall. It was able to co-evolve and maintain erroneous code that developers unnecessarily deleted. Generated tests for original and co-evolved code further showed that our co-evolution is not impacting the original code behavior w.r.t. the generated tests.

RQ3. How does our automatic approach for code co-evolution compare to the IDE quick fixes as a baseline?

The first baseline we compare to is the quick fixes available in the IDE since they are widely used by developers to repair code errors **10.1145/2384616.2384665**, and more importantly, they have not been compared to code co-evolution approaches before. To compare with the quick fixes as a baseline, we implemented the automatic application of quick fixes as an option in our plugin. It is a variant of our Algorithm 2 that only applies the quick fixes on the errors. The algorithm browses the errors of each compilation unit and applies the first proposed quick fix, since by default Eclipse IDE [5] order the quick fixes by *relevance*. Note that all executions herein terminated normally without any infinite loop.

In Table 4.9, we present the percentage of errors that quick fixes eliminated for each project, and the frequencies of each type of applied quick fixes.

While the quick fixes eliminated from 41% to 100% errors, we found that the precision and recall of automatic quick fixes are equal to 0, because no correction was applied as expected to the manual developers' resolutions. That is why we refer to the errors are eliminated by the quick fixes and not corrected or co-evolved. For example, concerning errors caused by class or property deletion from the metamodel, renaming a class or an

5. See *org.eclipse.jdt.ui.text.java.CompletionProposalComparator*

Table 4.9 – Number of applied Quick Fixes for each project and per evolved metamodel.

Evolved metamodels	Co-evolved projects	% of eliminated errors	N^o of applied Quick Fixes	Total
OCL Pivot.ecore	[$P1$]	82%	[Create Class X]: 3, [Create method m]: 27 , [Change m to m']: 60,[Add cast]: 59, [Change type (of variable or return type of method)]: 17, [Add unimplemented methods]: 43, [Create constants]: 15, [Remove argument]: 1.	225
	[$P2$]	41%	[Create method]: 4,[Change to m']:11, [change type of var or return type]: 1, [Add unimplemented methods]: 2,	18
Modisco Benchmark. ecore	[$P3$]	100%	[Create Class X]: 6, [Add unimplemented methods]: 1	7
	[$P4$]	100%	[Create Class X]: 2,[Create method]: 8, [Change m to m']; 5, [Remove argument]: 1, [Add cast] : 6	22
	[$P5$]	83%	[Create Method]: 17, [Change method m to m']: 16, [Remove argument]: 2, [Change type of var or return type]: 1, [Add Cast]: 16.	52
	[$P6$]	67%	[Change method m to m']: 4, [Add unimplemented methods]: 2, [Create Constant]:9, [change type] : 2, [Add Cast]: 6	23
Papyrus ExtendedTypes. ecore	[$P7$]	69%	[Create Class X]: 3, [Create method m]: 9, [Change method m to m']: 13, [Change type of var or return type]: 5, [Add Cast]: 4, [Add unimplemented methods]: 2.	36
	[$P8$]	93%	[Create Class X]: 1, [Create method] : 14, [Create Const] :3, [Add Cast]: 2, [Change method m to m']:3.	23
	[$P9$]	93%	[Create Class X]: 1, [Create method] : 14, [Create Constant] :3, [Add Cast]: 2, [Change method m to m']:3.	23
Total				429

attribute, moving, pushing, or pulling attributes or methods from a class to another, the quick fixes proposed to create them back in their old containers. This is in contradiction of the applied metamodel changes. For errors caused by changing a variable's type, the quick fixes always suggested adding a cast with a wrong type.

Unlike our approach, quick fixes do not take in consideration the context of the impacted code and the information contained in its causing metamodel changes. For example, the quick fix `create the missing method m()` is applied no matter the metamodel change (i.e., deletion, moving, pulling, or pushing changes) or the impacted code location (i.e., statement, variable declaration, parameter, etc.). Our approach takes into account the context of the impacted code and the causing change thanks to the use of pattern matching (see Section 4.2.5).

For instance, after a move property change, the resolution [$CR7$] or [$CR8$] is applied, respectively. Thus, taking into consideration the embedded information in the causing

Table 4.10 – Comparison between our automatic co-evolution approach and the semi-automatic co-evolution approach. [Legend: Precision(P) – Recall(R)]

Projects	P1	P2	P3	P4	P5	P6	P7	P8	P9	Total
Our automatic approach	90%(P) 66%(R)	80%(P) 83%(R)	100%(P) 100%(R)	81%(P) 75%(R)	58%(P) 58%(R)	48%(P) 51%(R)	98%(P) 94%(R)	93%(P) 100%(R)	93%(P) 100%(R)	82%(P) 81%(R)
Semi-automatic approach [11]	92%(P) 92%(R)	93%(P) 86%(R)	100%(P) 100%	100%(P) 100%(R)	100%(P) 100%(R)	48%(P) 48%(R)	100%(P) 100%(R)	100%(P) 100%(R)	100%(P) 100%(R)	92%(P) 91%(R)

metamodel change, namely the origin class, the target class, and the multiplicity of the reference between them.

***RQ*$_3$ insights:** The automatic application of quick fixes leads to a faulty evolved code. The same type of quick fixes is applied regardless of the context of the impacted code. With 0 precision and recall, we can conclude the inability of Eclipse quick fixes to co-evolve correctly the code with the evolution of the metamodel.

RQ4. How does our automatic approach for code co-evolution compare to the state-of-the-art semi-automatic approach as a baseline?

We now compare our approach to the state of the art of metamodel and code co-evolution, namely our previous work [11]. It runs an impact analysis to trace the impacts in the code, while we rely on the compilation errors after regenerating the code API from the evolved metamodel. We also consider an additional change of Pull property and add a new resolution for it ($CR14$). We further distinguish with our previous work by checking behavioral correctness with tests (before/after co-evolution) and by comparing to two baselines. Yet, the main difference is it being semi-automatic requiring intervention to decide between alternative resolutions to co-evolve the code, while we fully automate this decision based on our pattern matching process. Thus, intuitively, it will likely perform better than a fully automatic co-evolution. RQ4 aims to measure this difference.

Table 4.10 shows precision and recall values of our approach and the semi-automatic co-evolution approach [11]. We notice that the semi-automatic co-evolution approach [11] outperforms our automatic co-evolution approach by a small margin in terms of precision and recall. On average, we had 82% precision and 81% recall in full automatic co-evolution compared to 92% precision and 91% recall in semi-automatic co-evolution. This is due to the fact that our approach favors the least deletion when possible [$CR1$] over [$CR3$], or [$CR4$] over [$CR2$], while the semi-automatic approach favors integral deletions [$CR3$]

over [$CR1$], or [$CR2$] over [$CR4$]. Overall, we observe that the trade-off between full and semi-automatic co-evolution leads to only a reduction of -10% in precision and recall while still maintaining them at a high level ($> 81\%$). This small reduction in performance provides a bigger benefit of speeding up the co-evolution and removing the burden of manual intervention from developers on every code error. Indeed, rather than spending time in guiding the co-evolution at every error, developers can automatically perform it in couples of minutes, in particular, if thousands of errors must be co-evolved after metamodel evolution.

***RQ*$_4$ insights:** Fully automating metamodel and code co-evolution performs less than a semi-automatic co-evolution, but still gives a high precision and recall of ($> 81\%$) while removing the burden of manual intervention from developers on every code error to co-evolve.

4.3.5 Discussion and limitations

This section discusses the approach and the observed results.

The high-level intuition behind our approach is that we apply a targeted co-evolution by propagating the impacting metamodel changes only to the code errors directly, rather than browsing all code elements statically to fetch the possible impacts. Our automatic co-evolution approach tries to mimic the developers' behavior with known resolutions when co-evolving the code errors in an "iterative way". After the application of each resolution, the list of errors is refreshed to get the list of the new compilation errors for the next iteration.

Moreover, Table 4.6 shows the distribution of applied resolutions. It is related to the distribution of the impacting metamodel changes (see Table 4.1) since for a given impacting change, a specific set of resolutions can be applied. However, there is no relation between the distribution of the applied resolution and the recall and precision, which depends more on 'if' the automatic resolutions match "or not" the manual developer's expected resolutions. Moreover, we did not randomly and synthetically choose metamodel changes and their types in the evaluation. Rather, we took the realistic developers' evolutions of the metamodels in our selected real-world software projects. Thus, we did not influence the applied resolutions, which correspond only to the metamodel changes. Of course, having a different distribution of metamodel changes would result in a different distribution of applied resolutions.

Finally, we handled the evolution of one metamodel at once to co-evolve its impacts. Handling multiple metamodels is an interesting case, but is out-of-scope in this thesis. Nonetheless, the first favorable scenario we can currently handle is to treat the multiple metamodels in sequence when they are independent of each other. The main limitation is in a second scenario where there are dependencies between the multiple metamodels. For example, a change "set type to T" in a metamodel A referencing a change "add T" in a metamodel B. Herein, the solution would be to find the order of metamodels in which to handle the co-evolution in sequence. This is left as future work.

4.4 Threats to Validity

This section discusses threats to validity [220].

4.4.1 Internal Validity.

To provoke the code errors, we had to replace the original metamodel evolution and to regenerate the code API. To reduce any bias in the source of errors, we decided to evolve one metamodel at a time. This increases the confidence in the source of the errors, and hence, their co-evolution. Dealing with conflicting errors that can mask each other due to the simultaneous evolution of several metamodels is left for future work. Moreover, to measure the correctness, we analyzed the developers' manual co-evolution. To reduce the risk of misidentifying an expected resolution, for each impacted part of the code, we investigated the entire co-evolved class. If we did not find it, we further searched in other classes in case the original impacted part of the code was moved into another class. Thus, our objective was to reduce the risk of missing any correspondence between an error in the original code and its evolved version. Moreover, as our co-evolution relies on the quality of detected metamodel changes, we analyzed each detected change and checked whether it occurred between the original and evolved metamodels. This alleviates the risk of relying on an incorrect metamodel change that would degrade the pattern matching. Note that the order of changes taken as input does not influence our co-evolution, but the order of errors we treat may affect the distribution of the applied resolutions. However, we took the order in which the errors were detected. Finally, besides the manual checking of the co-evolved code, we used EvoSuite as a test suites generation tool that has shown its efficiency in test generation as a state-of-the-art tool [221], [222]. The main reason is that our projects did

not have manually written tests. However, automatic test generation can even be more advantageous herein. Indeed, it generates tests for all public methods, whereas developers tend to manually write few tests for only some targeted methods. Thus, if relying only on manually written tests, there is a high risk of not assessing the behavioral correctness of many cases of code co-evolutions that are not covered by test cases. Despite the fact that generated tests can be better for our approach and its results allow us to measure the behavioral correctness of the co-evolution, it still can be combined with manual written tests.

4.4.2 External Validity.

We implemented and evaluated our approach for EMF and Ecore metamodels and Java code. Although co-evolution could theoretically be applicable for other languages, such as C# or C++, we cannot generalize our results. Further experimentation on other languages is necessary. However, the only requirement to apply our approach to other languages is to parse the ASTs of the erroneous code and to adapt our resolutions to the new ASTs' structure.

Furthermore, the evaluation was carried out on Eclipse projects in three languages. Thus, we cannot generalize our findings to other software languages and their implementations. However, our approach could be used to co-evolve Java code added on top of a generated code from a domain model, *e.g.,* from a UML class diagram or an entity model. Nonetheless, more evaluations are still necessary.

4.4.3 Conclusion Validity.

Our evaluation showed promising results with an automatic code co-evolution that is fast and useful, with an average of 82% precision and 81% recall varying from 48% to 100%. The results also showed the usefulness of our approach and the resolutions proposed in our catalog in Table 4.1. However, even though we evaluated it on nine projects with complex metamodel evolution, we plan to further evaluate on more case studies to have more insights and statistical evidence.

4.5 Conclusion

In this chapter, we presented an approach to automatically co-evolve code when metamodels evolve. It relies on a pattern matching algorithm to match the different pattern usages of the metamodel generated code elements in the additional erroneous code. Once a direct error in the code is matched with a pattern usage, we can co-evolve it with its corresponding resolution. For indirect errors that cannot be matched, we apply the available quick fix to repair them. Our code co-evolution was evaluated on nine projects and three metamodels from three Eclipse EMF implementations. After re-generating the code API from the evolved version of the metamodel, hence, causing 837 errors in the additional code. For the 771 direct errors, our automatic co-evolution approach was able to match 631 pattern usages and to apply 631 resolutions. It showed to be efficient and useful in co-evolving all direct errors in the code with an average of 82% precision and 81% recall varying from 48% to 100%. All 66 indirect errors were resolved by calling 5 quick fixes. Moreover, we checked the behavioral correctness of our approach by using test suites before and after code co-evolution. We observed that the percentage of passing, failing, and erroneous tests remained stable with insignificant variations in some projects. Thus, suggesting the behavioral correctness of the co-evolution. Finally, we found that the quick fixes are not able to correctly co-evolve the code. This is because they cannot exploit the context of the impacted code and relate it to the changes of the metamodel. We also found that fully automating metamodel and code co-evolution performs less than a semi-automatic co-evolution, but still gives a high precision and recall of ($> 81\%$) while removing the burden of manual intervention from developers on every code error to co-evolve.

In summary, our work presents the following contributions : 1/ A fully automatic code co-evolution approach due to Ecore metamodel evolution based on pattern matching. 2/ A prototype implementation of an Eclipse plugin to show the efficiency of our approach. 3/ An evaluation based on four actions: 1) Measuring the co-evolution correctness. 2) Verifying the behavioral correctness using unit tests running before and after automatic code co-evolution. 3) Comparison with the state-of-the art semi-automatic co-evolution approach [11]. 4) Comparison with Quick Fixes popular tool.

Chapter 5

Automated testing of metamodels and code co-evolution

Whether developers co-evolve their code either manually or automatically, they cannot ensure that the code co-evolution is behaviorally correct, i.e., without altering the behavior of their impacted code. Especially, when there is other alternative co-evolutions for the same impacted code (challenge Ⓒ2). One way to check this is to re-run all the tests after each code co-evolution, which is expensive and time-consuming. It is also tedious and error-prone when the developer checks the output of the tests' execution and manually maps them between the original and evolved versions.

This chapter presents a new fully automatic approach to check the behavioral correctness of the code co-evolution between different releases of a language when its metamodel evolves. This approach leverages the test suites of the original and evolved versions of the language, and hence, its metamodels and code.

Section 5.1 presents the key concepts as a background followed by a motivation example. In section 5.2, I detail the approach for testing metamodel and code co-evolution and how I use unit test suites before and after code co-evolution to check that the co-evolution did not alter the behavior of the code, while Section 5.3 evaluates it. The evaluation section presents first our user study experiment to gain evidence on the difficulty or not of the manual task of tracing impacted tests after metamodel evolution and co-evolution. Then, I measured the efficiency of this approach and the gains observed compared to using the whole test suite as a baseline. Sections 5.4 discusses internal and external threats to validity. Finally, Section 5.5 concludes the chapter.

5.1 Background and Example

This section gives a background on how metamodels play a significant role when building software languages and their tooling for a better comprehension of the current

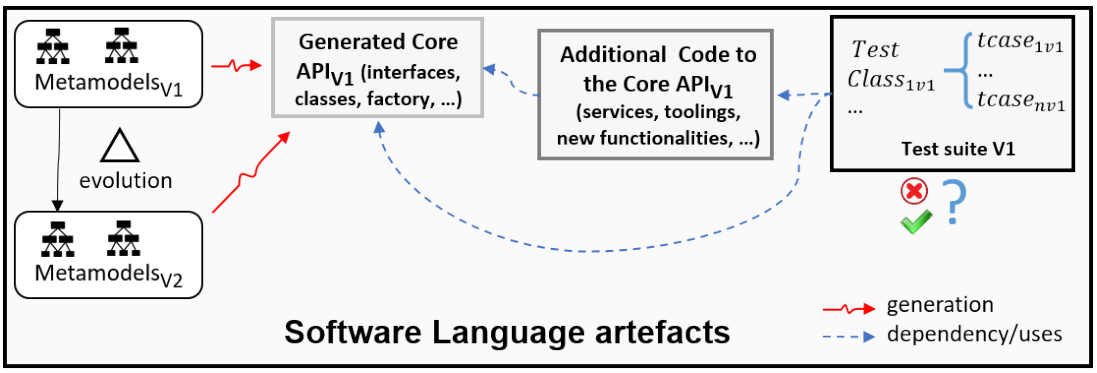


Figure 5.1 – Evolution of metamodels and related artifacts of a software language

work. It then discusses the scenario of co-evolution that arises between metamodels and code, and the need for testing its behavioral correctness.

5.1.1 Key Concepts

As I already explained in Section 1.1, once metamodels are defined and validated, core API code is generated, developers aim to enrich it by implementing method bodies and adding advanced functionalities, such as validation services and execution engines. After that, a test suite is added on top of the generated and the additional code to test the implemented functionalities, as illustrated in Figure 5.1. This can be done manually or with the existing techniques for automated test generation [158], [224], [225].

However, the generated code, additional code and tests hold for a single version of the metamodel. With metamodel evolution comes the challenges of co-evolution and its correctness. For example, when a metamodel evolves, model instances must be co-evolved. One way to check the models' correctness is to rely on the OCL constraints to verify the static semantic of the models [3], [226]. Thus, one can compare the constraints before and after the models' co-evolution. In our case, when the metamodel evolves, the API code can be re-generated again. As a consequence, the additional code manually integrated by developers must be co-evolved accllmordingly as well. Unfortunately, there is also no guarantee that the code co-evolution is correct. Usually, the test suite is used to identify possible bugs in the new evolved version of the code. In this work, similarly to the practice of regression testing, we leverage the test suites in both the original and evolved versions of the code to check particularly the behavioral correctness of the co-evolution.

Indeed, in regression testing, the goal is to re-run tests after any code changes to ensure that the software still works as intended [227]–[229]. In this contribution, we intend to

follow a similar methodology by tracing the impacted tests that must be re-run to compare their results before and after code co-evolution.

5.1.2 Motivating Example

This section introduces a motivating example to illustrate the challenge of metamodel and code co-evolution and testing it.

Figure 4.2,Page 57 shows an excerpt of the "Modisco Discovery Benchmark" metamodel [1] consisting of 10 classes in version 0.9.0. It illustrates some of the domain concepts `Discovery`, `Project`, and `ProjectDiscovery` used for the discovery and reverse engineering of an existing software system. From these metaclasses, a first code API is generated, containing Java interfaces and their implementation classes, a factory, a package, etc. In version 0.11.0, the Modisco metamodel evolved with several significant changes, among which we find: 1) Renaming the property *totalExecutionTimeInSeconds* to *discoveryTimeInSeconds* in metaclass `Discovery`, followed by 2) Moving the property *discoveryTimeInSeconds* (after its renaming) from metaclass `Discovery` to `DiscoveryIteration` as illustrated in Figure 4.2, Page 57.

Listing 5.1 shows a directly impacted test from the class `DiscoveryImpl_ESTest` after the evolution of Modisco metamodel. It tests directly the evolved method. In the class `DiscoveryIterationImpl_ESTest` of Modisco 0.11.0, the test shown in Listing 5.2 is impacted indirectly by the same change. Indeed, the method *totalExecutionTimeInSeconds* after its rename and move is used in the method *eSet* as shown in Listing 5.3, which is in turn used in the unit test shown in Listing 5.2. It tests indirectly the evolved method.

The above examples show the direct and indirect impact of the metamodel evolution on the code and on the tests. However, manually tracing the impact of multiple metamodel evolutions at once till the test is tedious, error-prone and time-consuming. In particular, this tracing must be done before and after the metamodel evolution and then to map the traced tests to investigate the code co-evolution correctness.

The next section presents our contribution for an automated tracing of the tests impacted by the evolution of the metamodel that allows later to check the behavioral correctness of the metamodel and code co-evolution.

Listing 5.1 – Excerpt of a directly impacted test in Modisco.

1. `https://git.eclipse.org/r/plugins/gitiles/modisco/org.eclipse.modisco/+/refs/tags/0.12.1/org.eclipse.modisco.infra.discovery.benchmark/model/benchmark.ecore`

```
@Test(timeout=4000)
public void test000() throws Throwable {
    DiscoveryIterationImpl discoveryIterationImpl0 = new DiscoveryIterationImpl();
    ...
    assertFalse(discoveryIterationImpl0.eIsProxy());
    assertEquals(0.0,  (*\ul{discoveryIterationImpl0.getDiscoveryTimeInSeconds()}*),
        0.01);
    assertTrue(discoveryIterationImpl0.eDeliver());
    assertEquals(0.0, discoveryIterationImpl0.getSaveTimeInSeconds(), 0.01);
    assertEquals(0L, discoveryIterationImpl0.getMaxUsedMemoryInBytes());
    assertTrue(discoveryIterationImpl0.eDeliver());
    ...
}
```

Listing 5.2 – Excerpt of an indirectly impacted test in Modisco.

```
@Test(timeout = 4000)
public void test16()  throws Throwable  {
    DiscoveryIterationImpl discoveryIterationImpl0 = new DiscoveryIterationImpl();
    EList<Event> eList0 = discoveryIterationImpl0.getMemoryMeasurements();
    try {
        discoveryIterationImpl0. (*\ul{eSet(30, (Object) eList0)}*);
        fail("Expecting␣exception:␣ClassCastException");

    } catch(ClassCastException e) {
        verifyException("org.eclipse.modisco.infra.discovery.benchmark.impl.
            DiscoveryIterationImpl", e);
    }
    assertEquals(0.0, discoveryIterationImpl0.getSaveTimeInSeconds(), 0.01);
    ...
}
```

Listing 5.3 – Excerpt of an impacted method in Modisco.

```
@Override
public void eSet(int featureID, Object newValue) {
    switch (featureID) {
        ...
        case BenchmarkPackage.DISCOVERY_ITERATION__DISCOVERY_TIME_IN_SECONDS:
        (*\ul{setDiscoveryTimeInSeconds((Double)newValue)}*);
        return;
        case BenchmarkPackage.DISCOVERY_ITERATION__SAVE_TIME_IN_SECONDS:
        setSaveTimeInSeconds((Double)newValue);
        return;
        case BenchmarkPackage.DISCOVERY_ITERATION__MAX_USED_MEMORY_IN_BYTES:
        setMaxUsedMemoryInBytes((Long)newValue);
        return;
        ...
    }
```

5.2 Approach

This section presents our proposed overall approach. It first gives an overview. Then, it describes how to detect the metamodel changes and how to trace their impacts until the tests and map them. Finally, it details our prototype implementation.

5.2.1 Overview

The overall objective of our approach is to help developers in checking the behavioral correctness of the code co-evolution when metamodels evolve, as the co-evolution may be done incorrectly or in an incomplete way (i.e., referred to as partial co-evolution in [230], [231]). Several ways exist, such as using formal methods, manual code review, or unit tests, etc. Our scope lies in tracing the impact of the metamodel changes till the tests and rely on them as an indicator for behavioral correctness of the code co-evolution, similarly as in a regression testing method [227]–[229]. Our vision is rather than letting the developers execute all test suite in both versions and manually analyzing them, we can reduce the set of tests to be analyzed to the only minimum necessary one. Thus, saving effort and time for developers.

Figure 5.2 depicts the overall approach workflow. We first compute the difference between the two metamodel versions each of them having a generated code and an additional code (step 1). In the original version, the additional code is the impacted one, and in the evolved version, the additional code is the co-evolved one. After that, we run the impact and the test tracing analysis to link the metamodel changes to the impacted and co-evolved code and their respective tests (step 2). Therefore, a developer can run the traced tests before and after the code co-evolution to check their behavioral correctness. Finally, to ease this task, we map the traced tests and execute them to report them back in a form of a diagnostic to the developers for an easier in-depth analysis of the effect of metamodel evolution rather than analyzing the whole test suite (step 3). Therefore, in a nutshell, there are no particular preconditions to our approach except having available code and tests from both before and after co-evolution along with the delta of the metamodel changes.

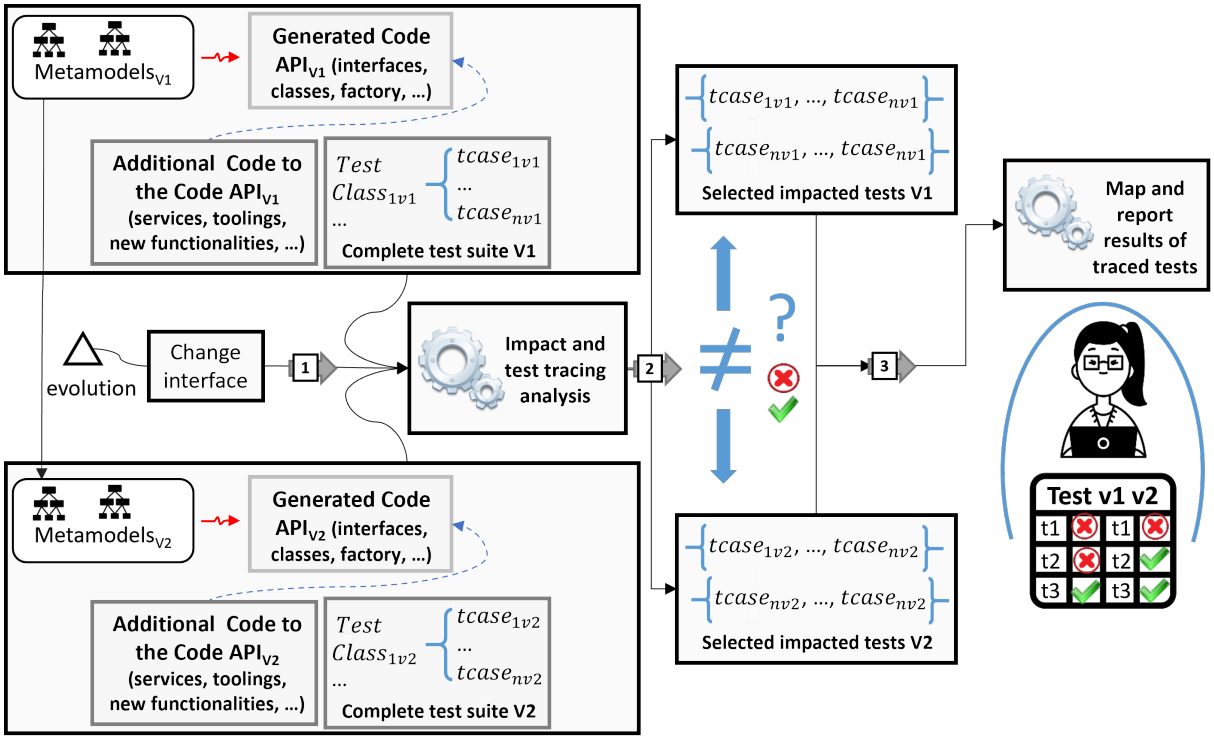


Figure 5.2 – Overall approach

5.2.2 Detection of metamodel Changes

As described in Chapter 4.2.2, we use an interface specification of changes [1] that is a connection layer to our test tracing approach with the existing change detection approaches [59], [61]–[65].

In practice, we focus on the impacting metamodel changes that will require co-evolution of the code and not on the non-impacting changes. For example, a delete change or a change of type will impact the code and possibly its behavior that can be observed with its tests. However, addition changes, although non-breaking, can be traced back to their newly added tests. Thus, we also consider them to observe their behavior. The list of metamodel changes [63], [209] we consider for tracing their impact up to the tests is as shown in the first column of Table 5.1.

For each version of the tests, we use different information provided by each metamodel change, depending on whether we are tracing the impacted tests in the original or the evolved versions. Columns 2 and 3 of table 5.1 detail the treatments of each metamodel change in the original and evolved versions. For example, for a rename element *e* to *e'*, we search for *e* and *e'*, respectively, in the original and evolved versions. Similarly, for

Table 5.1 – List of metamodel changes and how they are traced up to the tests in the original and evolved versions.

Metamodel changes	Tests treatment	
	In original version (V1)	In evolved version (V2)
◇ Delete property p in class `C`	Search for usages of p in `C`	*n/a*
◇ Delete class `C`	Search for usages of `C`	*n/a*
◇ Add property p in class `C`	*n/a*	Search for usages of$p$ in `C`
◇ Add class `C`	*n/a*	Search for usages of `C`
◇ Rename element e to e' in class `C`	Search for usages of e in `C`	Search for usages of e' in `C`
◇ Change multiplicity of property p in class `C`	Search for usages of p in `C`	
◇ Change type of property p from `S` to `T`	Search for usages of p in `C`	
◇ Move property p_i from class `S` to `T` through *ref* ◇ Extract class of properties p_1, ..., p_n from `S` to `T` through *ref*	Search for usages of all p_i in `S`	Search for usages of all p_i in `T`
◇ Push property p_i from class `Sup` to $\texttt{Sub}_1$,...,$\texttt{Sub}_n$	Search for usages of all p_i in `Sup`	Search for usages of all p_i in all $\texttt{Sub}_i$
◇ Pull property p_i from classes $\texttt{Sub}_1$,...,$\texttt{Sub}_n$ to `Sup`	Search for usages of all p_i in all $\texttt{Sub}_i$	Search for usages of all p_i in `Sup`
◇ Inline class `S` to `T` with properties $p_1, ..., p_n$	Search for usages of all p_i in `S`	Search for usages of all p_i in `T`

the other changes, such as Move, Pull, Push, etc. where the source and target classes are different in the original and evolved versions. Only the impact of delete changes is searched in the original version, while the impact of addition changes is only searched in the evolved version.

5.2.3 Tracing the Impacted Tests

Our approach traces the impact of metamodel changes up to the test. To do that, we structure the code source to better navigate in it. Before starting, we parse the code source including tests and build the Code Call Graph (CCG) at the methods level. It consists of nodes $\mathbb{N}$ that are methods, and edges $\mathbb{E}$ that are calls between methods. For a given method, the CCG allows us to retrieve its callers, hence, tracing the call methods recursively up to the tests. After that, we apply Algorithm 1 on the built CCG. Overall, for each detected metamodel change, the algorithm computes the list of direct and indirect impacted tests that can be traced to the given metamodel change.

First, we analyze the AST of the code to identify the code usages of the evolved

metamodel element. The information concerning the metamodel element before and after evolution, which is included in the metamodel change (see Table 5.1), allow us to spot the impacted code usages (Line 1). For example, for a rename property *id*, the algorithm will first find its usages, such as *getId()* or *setId()*[2]. Then, we filter these impacted code usages by keeping only the ones found inside a method declaration. Let us call the found method declaration using the impacted code *IM()* (Line 5). If *IM()* is a test method, that means that we found a direct impacted test (Line 7). Otherwise, thanks to the CCG, we retrieve *IM()*'s parents *parentsOfIM*, which are all the method declarations invoking IM() (Line 9-17). Afterwards and recursively, we check each parent of *IM()* if it is a test method or not (Line 14). The process is finished if we reach a method declaration that has no parents in the CCG that is either a test or not, or if the reached method declaration is already treated in the *parentsOfIM*. Therefore, by design, after browsing all the impacted code usages, Algorithm 1 traces the list of all impacted tests, without missing any if impacted. It is worth noting that tracing all impacted tests holds syntactically and w.r.t. static semantics. Possible side-effect will require further advanced dynamic analysis and is left for future work. Listing 5.1 presents an example of an impacted test in the Eclipse Modisco.infra.discovery.benchmark project. As described in Section 5.2.2, we detect that the attribute setTotalExecutionTimeInSeconds is renamed and moved from the class Discovery to the class DiscoveryIteration. After that, Algorithm 1 detects the code usage *getDiscoveryTimeInSeconds*. Then, it traces it to the method *test000*. As it has the *@Test* annotation, we conclude that *test000* is an impacted test due to the detected move change. Note that a test can be impacted by multiple metamodel changes, and one metamodel change can impact many tests. Algorithm 1 will detect either cases.

5.2.4 Mapping of impacted tests

After having traced the impacted tests, we further assist developers in analyzing the output of our approach. We provide a diagnostic in a form of a visualized report. This report displays the mapping of impacted tests between the original and evolved versions, along with the verdict of their execution (i.e., pass, fail, and error) and the corresponding impacted change. As an input to generate the diagnostic, the developer selects two impacted traced test classes in the original and evolved versions. The mapping

2. Note that the knowledge about the generated code elements from the metamodel elements (e.g., getter/setter for EAttribute, class/interface for EClass, etc.) is so far hard-coded in the implementation of Algorithm 1. The mappings must be provided for our approach to be able to trace the tests.

between the two sets of test cases for these classes is performed using a state-of-the-art tool, namely GumTree [232]. It parses both test classes into a tree structure to enable the matching of the test cases. Herein, we distinguish three cases, namely: 1) tests that exist in both versions, 2) tests that exist in the original version but not in the evolved one, and 3) tests that exist in the evolved version but not in the original one. The impacted tests are then executed programmatically using JUnit runner. To facilitate the analysis and the tracing of impacted tests, we include the corresponding impacting metamodel changes in an additional column of the diagnostic report.

Algorithm 1 Impacted tests detection

```
Require: codeCallGraph, change
impactedUsages ← match(AST, change)
impactedTests ← ϕ for (impactedUsage ∈ impactedUsages ) do
/* Find the method declaration using impactedUsage*/
IM ← getIM(impactedUsage, codeCallGraph) if (isTest(IM)) then
impactedTests.add(IM) /*If not already added*/
    else
parentsOfIM ← getParents(IM, codeCallGraph)
nextRound.add(parentsOfIM)
    while (nextRound.hasNewIMs()) do
IM ← nextRound.get()
    if (isTest(IM) then
impactedTests.add(IM)/*If not already added*/
    else
parentsOfIM ← getParents(IM, codeCallGraph)
nextRound.add(parentsOfIM)
```

Figure 5.3 illustrates a screenshot of the test tracer report on a toy example "Employee Management Project". After selecting the class of tests that have been traced before code co-evolution, and the class of tests that have been traced after code co-evolution, the user clicks on "Map tests" to display the table of mapped tests with their verdict of execution. To illustrate the verdict of the test execution, we made: the passing test in green, the failing test in blue, and erroneous tests in red. For example, the change "Delete Class Contact" impacts two tests, test11 which passes, and test06 which fails. The verdict of the execution of the tests has no relation with the change itself but with the test that uses the code elements impacted by the metamodel change. Those tests do not exist anymore in the evolved version since the class Contact is absent and cannot be tested anymore.

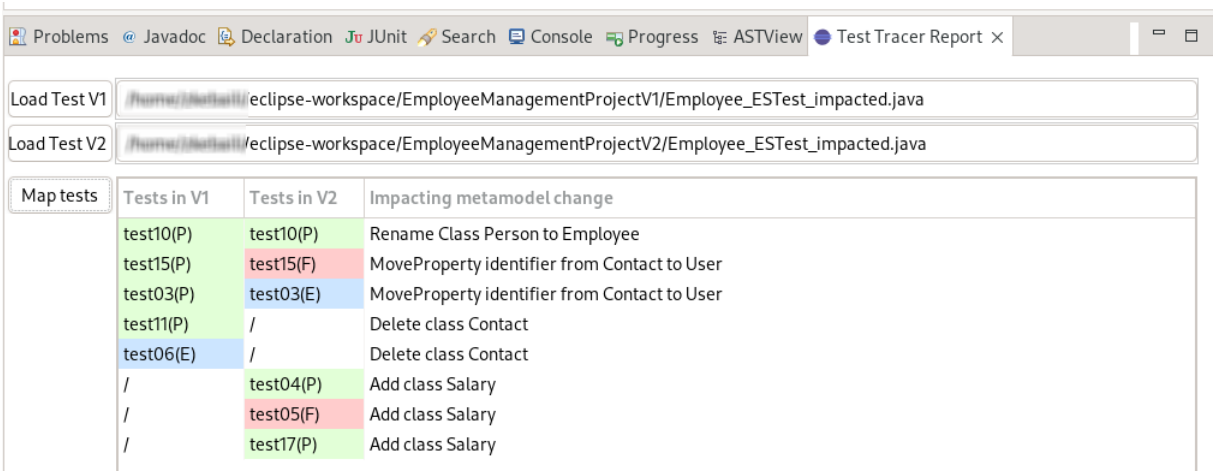


Figure 5.3 – A snippet of the diagnostic report view to visualize and analyze the traced impacted tests.

5.2.5 Tool implementation

We implemented our solution as an Eclipse Java plugin handling Ecore/EMF metamodels and their Java code. We rely on our approach [60] to perform the detection of the metamodel changes bridged with the change interface. The test tracing, technically, consists of parsing the java code and manipulating its AST using JDT eclipse plugin [3] to construct the Code Call Graph (CCG). After that, we navigate within the methods calls until we either reach a test or not. Finally, for each impacted *TestClass*, we create a copy *TestClass_Impacted* where we only include the impacted traced test cases. The developer can then launch the traced tests in both the original and evolved versions to investigate the behavioral correctness of the co-evolved code. We further implemented the diagnostic report as a view that shows the mapped tests with GumTree [232], their verdict retrieved programmatically using JUnit Runner [4] and impacting metamodel changes, as shown in Figure 5.3. The goal is to ease the developers' in-depth analysis of the effect of metamodel evolutions rather than rerunning and analyzing the whole test suite.

5.3 Evaluation

This section evaluates our automatic approach of checking the behavioral correctness of the metamodel and code co-evolution. First, we present the data set and the evaluation

3. Eclipse Java development tools (JDT): `https://www.eclipse.org/jdt/core/`
4. `https://junit.org/junit4/javadoc/4.13/org/junit/runner/Runner.html`

Table 5.2 – Details of the metamodels and their evolutions.

Evolved metamodels	Versions	Atomic changes in the metamodel	Complex changes in the metamodel
Pivot.ecore in project *ocl.examples.pivot*	3.2.2 to 3.4.4	*Deletes:* 2 classes, 16 properties, 6 super types *Renames:* 1 class, 5 properties *Property changes:* 4 types; 2 multiplicities *Adds:* 25 classes, 121 properties, 36 super types	1 pull property 2 push properties
Pivot.ecore in project *ocl.pivot*	6.1.0 to 6.7.0	*Deletes:* 0 classes, 4 properties, 4 super types *Renames:* 0 class, 1 properties *Property changes:* 49 types; *Adds:* 5 classes, 47 properties, 7 super types	n/a
ExtendedTypes.ecore in project *papyrus.infra.extendedtypes*	0.9.0 to 1.1.0	Deletes: 10 properties, 2 super types Renames: 3 classes, 2 properties Adds: 8 classes, 9 properties, 8 super types	2 pull property 1 push property 1 extract super class
Benchmark.ecore in project *modisco.infra.discovery.benchmark*	0.9.0 to 0.13.0	Deletes: 6 classes, 19 properties, 5 super types Renames: 5 properties Adds: 7 classes, 24 properties, 4 super types	4 moves property 6 pull property 1 extract class 1 extract super class
Ecore.ecore in project *org.eclipse.emf*	2.37.0 to 2.37.0'	Deletes: 1 class, 2 properties Renames: 2 properties	1 move property 1 pull property

process. Then, we set the research questions we address and discuss the obtained results.

5.3.1 Data Set

This section presents the used data set in our evaluation to be found in the attached supplementary material [5].

We evaluate our approach on four case studies of language implementations in Eclipse, namely OCL [218], Modisco [204], Papyrus [219], and EMF [233] project. OCL is a standard language defined by the Object Management Group (OMG) to specify First-order logic constraints. Modisco is an academic initiative to support development of model-driven tools, reverse engineering, verification, and transformation of existing software systems. Papyrus is an industrial project led by CEA [6] to support model-based simulation, formal testing, safety analysis, etc. The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model [223]. Thus, the four case studies cover standard, academic, and industrial languages that have evolved several times for more than 10 years of continuous development period.

5. `https://figshare.com/s/b6251b9e47fa82983ce5`
6. `http://www-list.cea.fr/en/`

Moreover, we aimed at selecting meaningful evolutions that do not consist in only deleting metamodel elements, but rather include complex evolution changes. We also aimed to select long and short evolution intervals in the selected releases versions to stress test our approach in different scenarios. This is the case for the OCL, Modisco, and Papyrus case studies. However, they do not have manually written tests. Thus, we added the EMF case study that have manually written tests, but its metamodel is stable with no evolutions. Therefore, we had to simulate a set of metamodel evolution changes similar to those real-world changes observed in our three first case studies and we co-evolved their impacts with our previous work [11].

Table 6.3 gives details about the selected case studies, in particular about their metamodels and the changes applied during evolution. The total of applied metamodel changes was 452 atomic changes and 21 complex changes in the five metamodels. Table 6.4 gives details on the size of the projects in terms of code and tests of the original and evolved versions. We collected a total of 18 projects to evaluate our approach on.

5.3.2 Evaluation Process

We evaluate our approach by: 1) investigating its usefulness compared to the manual tracing of the impacted tests with user study, 2) measuring its ability to automatically trace the impacted tests due to impacting metamodel changes both in the original and evolved version of the project, 3) assessing its ability to give an indication about the correctness of the code and metamodel co-evolution, and finally 4) measuring the gains of its usage in terms of reduction of tests and their execution time. Note that as the metamodel changes are taken as input of our automatic test tracing, we studied the original and evolved versions to confirm the metamodel changes. Therefore, we do not take an incorrect input of metamodel changes that would mislead our traced tests, which would mislead the behavioral checking of the code co-evolution.

Regarding the tests, only the EMF case study had manually written tests. Thus, we had to generate tests for OCL, Modisco, and Papyrus case studies. We used a state-of-the-art tool, namely EvoSuite [158]. EvoSuite is a search-based tool for Unit Tests generation. It uses a heuristic algorithm, particularly, genetic algorithm in the Test Suit generation. In their used approach Fraser et al. [158] aimed to maximize the coverage metric and mutation score which guarantees good-quality tests. EvoSuite is largely used and it was evaluated not only in literature but also in the industrial context. Firhard et al. [234] compared it with DSpot, a state-of-the-art tool for test amplification, their results show

that EvoSuite achieves a statistically better mutation score. Herculano et al. found that EvoSuite's generated tests can successfully help to identify faults during maintenance tasks [235]. In industry, Rozière et al. use automated tests generated with EvoSuite to filter invalid code translations in the context of their work done for Meta [185]. Gruber et al. [216] further showed the quality and robustness of the generated tests. It showed that while flakiness is at least as common in generated tests as in developer-written tests, EvoSuite is effective in alleviating this issue giving 71.7% fewer flaky tests. Thus, EvoSuite is appropriate in our work to generate robust tests in the original code and in the co-evolved code to compare their results, i.e., check behavioral correctness. We simply let EvoSuite run to generate Junit test classes for the selected projects with the following parameters: *-DmemoryInMB=2000 -Dcores=4 -DtimeInMinutesPerClass=10 evosuite:generate evosuite:export*. It uses up to 2GO of RAM, 4 CPU cores, and 10 minutes per test class. Generating tests is a best practice, in particular w.r.t. its efficiency in test generation and at a large scale for all public methods [221], [222]. EvoSuite generates tests for all public methods, whereas developers tend to manually write a few tests for only some targeted methods. Thus, relying only on manually written tests increases the risk of not assessing the behavioral correctness of many cases of code co-evolutions that are not covered by test cases. Generating tests alleviates this risk. Indeed, this is observed when computing the coverage metric for each of our considered projects. Table 5.4 shows that the highest coverage (69% to 95%) is obtained on projects with automatically generated tests and the lowest coverage (17% to 33%) were on the two projects with manually written tests.

5.3.3 Research Questions

This section sets the research questions (RQs) to assess our work. The research questions are as follows:

RQ0. *To what extent can developers manually trace the tests impacted by the evolution of the metamodel?* This aims to asses if developers can manually trace the tests that are impacted by the changes of the metamodel. This also aims further to assess our approach's usefulness through main observations.

RQ1. *To what extent does our automatic approach trace the impact of the metamodel evolution to the tests?* This aims to assess on real-world case studies the ability and applicability of our automatic approach to trace the metamodel changes with code elements till their tests.

Table 5.3 – Details of the projects and their tests.

Projects co-evolved in response to the evolved metamodels	N^o of packages	N^o of classes	N^o of test packages	N^o of test classes	N^o of LOC	N^o of tests
[$P1_{V1}$] ocl.examples.pivot	22	439	22	290	74002	7322
[$P1_{V2}$] ocl.examples.pivot	22	480	22	220	89449	4990
[$P2_{V1}$] ocl.examples.base	12	181	12	119	17617	2320
[$P2_{V2}$] ocl.examples.base	12	181	12	118	17596	2133
[$P3_{V1}$] ocl.pivot	60	1006	55	598	142236	8795
[$P3_{V2}$] ocl.pivot	63	1090	58	683	153613	6396
[$P4_{V1}$] papyrus.infra.extendedtypes	7	37	7	19	2057	135
[$P4_{V2}$] papyrus.infra.extendedtypes	7	51	7	26	2570	248
[$P5_{V1}$] papyrus.infra.extendedtypes.emf	5	25	4	14	1145	104
[$P5_{V2}$] papyrus.infra.extendedtypes.emf	5	25	4	14	1145	104
[$P6_{V1}$] papyrus.uml.tools.extendedtypes	5	15	3	9	726	75
[$P6_{V2}$] papyrus.uml.tools.extendedtypes	5	15	3	9	725	75
[$P7_{V1}$] org.eclipse.modisco.infra.discovery.benchmark	3	28	3	15	2333	524
[$P7_{V2}$] org.eclipse.modisco.infra.discovery.benchmark	3	30	3	15	2588	619
[P_{ecore_V1}] org.eclipse.emf.ecore	13	168	/	/	142586	0
[P_{ecore_V2}] org.eclipse.emf.ecore	13	166	/	/	141434	0
[$P8_{V1}$] org.eclipse.emf.test.core	/	/	19	141	40858	322
[$P8_{V2}$] org.eclipse.emf.test.core	/	/	19	141	40544	322
[$P9_{V1}$] org.eclipse.emf.test.xml	/	/	6	27	12088	64
[$P9_{V2}$] org.eclipse.emf.test.xml	/	/	6	27	12088	64

Table 5.4 – Coverage metric of each evaluation project.

Projects	[$P1$]	[$P2$]	[$P3$]	[$P4$]	[$P5$]	[$P6$]	[$P7$]	[$P8$]	[$P9$]
Coverage V1	66.9%	86.1%	80.1%	95.6%	95.1%	89.5%	91.3%	18%	33.4%
Coverage V2	66.2%	85.4%	74.1%	95.2%	95.2%	89.2%	87.5%	17.2%	33%

RQ2. *What is the observed behavioral correctness level of the code co-evolution?* This aims to assess through running the selected tests the effect of the code co-evolution, whether it keeps the tests' results stable, or instead degrade or improve them.

RQ3. *What are the observed gains (w.r.t. test case reduction and execution time) obtained from our approach of tracing impacted tests by metamodel evolution ?* This aims to highlight the benefit of our approach compared to when not using it and relying on the whole test suite as a baseline.

5.3.4 Results

We now discuss the results w.r.t. our research questions.

RQ0. To what extent can developers manually trace the tests impacted by the evolution of the metamodel?

The first goal of our evaluation is to gain evidence on the difficulty or not of the manual task of tracing impacted tests. Thus, we designed and ran a user study experiment.

RQ0 Set Up We first describe our user study experiment.

Subjects selection. The experiment was conducted with 8 participants (2 females and 6 males), including PhD students and research engineers in IRISA Laboratory, at the University of Rennes. The participants have a varying level of experience in programming (from 3 to 12 years with an average of 7 years), and a varying level in model-driven engineering (MDE) (from 0 to 7 years with an average of 2 years and 6 months).

Experiment Task. The experiment aims to evaluate the ability of participants to trace manually and analyze the affected unit tests before and after the metamodel evolution. We prepared two Eclipse workspaces. The first one contains the original version of the project `org.eclipse.emf.test.core`, and the second workspace contains the evolved version of the same project. The number of tests is 322 in both versions. The number of tests that must be traced is respectively 173 and 17 in the original and evolved versions. We then give in the guideline of the experiment the list of the changes that details the evolution of the metamodel (see last row of Table 6.3) with a description of the metamodel and project. We also explain what type of code elements are generated from each metamodel element. Each participant had then to identify impacted unit tests in the original and evolved version of the project `org.eclipse.emf.test.core`. This procedure not only highlights the direct impacts of the metamodel changes but also requires the consideration of indirect impacts. Additionally, the study explores the usefulness and potential adoption of our approach as an automatic support tool for tracing the impacted tests. After the end of this task, we presented our automatic tracing approach to the participant then we ran a post-questionnaire.

Variables. Our user study aimed to measure to what extent can developers trace impacted tests. The independent variable we controlled was the impacting metamodel changes. We covered seven different types of changes with both atomic and complex changes. We then observed the dependent variable of the traced tests by the participants.

RQ0 Results When we analyzed the answers of each participant, we found that they were able to trace only a few tests. In the original version of the project, the total number

the manually traced tests varied between 1 and 18, with an average of 6 tests out of the 173 impacted tests. In the evolved version of the project, the total number the manually traced tests varied between 2 and 32, with an average of 11 tests out of the 17 impacted tests. While we first observe that none of the participants traced all tests, they also did not correctly trace the tests.

Indeed, the number of correctly manually traced tests varies between 1 to 18 in the original version of the project, with an average of 5 traced tests. In the evolved version, the number of correctly manually traced tests varies between 0 to 10 tests with an average of 4 tests. We had five participants out of eight who wrongly traced eight tests in the original version of the projects, varying between one or two tests for each of them. In the evolved version of the project, all participants have wrongly traced between one to 26 tests that are not impacted by the evolution of the metamodel. We investigated the cause of this incorrect tracing. We found that one reason was the tracing of tests that contain a commented impacted code. Another reason was traced the wrong overload method `getEEnumLiteral(EInt)` of the actually evolved method `getEEnumLiteral(EString)`. Another reason was to simply include sibling tests in the class of a trace test.

In addition, we found that five participants considered both the direct and indirect impact of the metamodel evolutions on the tests, while the 3 remaining participants considered only the directly impacted tests. The results of the user study show the difficulty of manually tracing the tests with the evolved metamodel. Not only the participants could not trace all tests, but they even wrongly traced non-necessary tests.

Regarding the answers of the participants about usefulness [7] of our approach as an automatic support tool for tracing the impacted tests. One participant graded our approach as 'Somewhat useful', five out of eight graded it as 'Very useful', and two graded it as 'Extremely useful'. The last question was about their potential adoption [8]of our approach as an automatic support tool for tracing the impacted tests. Two participants out of eight answered 'Somewhat likely', three participants answered 'Likely', and the three remaining participants answered 'Very likely'. The finding of this experiment emphasizes the adoption likelihood and usefulness of our approach (discussed in RQ4).

7. Between 'Useless – Little useful – Somewhat useful – Very useful – Extremely useful'.
8. Between 'Very unlikely - Unlikely - Somewhat likely - Likely - Very likely'.

***RQ*$_0$ insights:** From our user study experiment, we observe that tracing manually the tests impacted by the evolution of the metamodel is a hard and error-prone task. The post-questionnaire results after a demonstration of our automatic approach suggest its high usefulness and adoption likelihood.

RQ1. To what extent does our automatic approach trace the impact of the metamodel evolution to the tests?

Following the evaluation protocol, we executed our approach on our case studies in Table 6.4. Figure 5.4 depicts the number of traced tests due to the impacting metamodel changes in Table 6.3. We first observe that we can trace tests successfully. We traced a total of 1608 out of 34612 tests due to 473 metamodel changes, distributed in 1106 and 502 tests in the original and evolved versions. Thus, we can isolate for the developers the tests that must be executed and looked at to check the behavioral correctness of the co-evolution.

We also observe that the more the number of evolution changes between the original and evolved versions of the metamodel increases, the more the number of tests we trace increases as well. In particular, when a lot of tests are available for analysis. This is true for the OCL Pivot project [$P1$] and Modisco project [$P1$]. We also observe no overall difference between projects with automatically generated tests and projects with manually tests. We traced similar ratios of tests.

Moreover, as several deletions of classes and properties occurred in the evolution changes, several tests are not generated in the evolved version, which explains why we trace more tests in the original version than in the evolved version. This is observed in most of the projects except the [$P4$], where [$P4_{V2}$] had more tests. We double-checked this case, and we found that there were strangely more tests generated by EvoSuite in [$P4_{V2}$] than in [$P4_{V1}$] for the same classes, likely due to more dependencies available in V2.

Finally, regarding the overhead, the time performance varied from 5 minutes in projects Papyrus Extendedtypes with 42 metamodel changes and 726 LOC up to 60 minutes in projects OCL Pivot with 117 metamodel changes and 142236 LOC. This is of course to be compared to manual tracing of the tests in both original and evolved versions before and after co-evolution, which can be tedious and time-consuming. In particular, when thousands of tests exist as in the project of OCL Pivot. However, our prototype traces the impact of the metamodel changes sequentially as a proof-of-concept for feasibility and applicability. Time performance can further be improved by parallelizing the tracing for

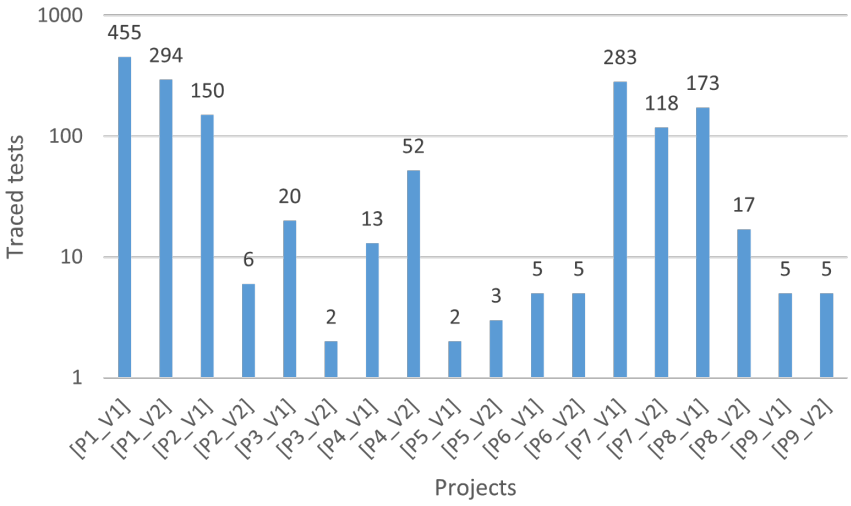


Figure 5.4 – Traced tests due to metamodel evolution in each project.

the metamodel changes. This is left as future work.

> ***RQ*$_1$ insights:** We could successfully trace the tests that must be executed before and after the co-evolution regardless of whether they are manually or automatically written. This would help developers to immediately check the code co-evolution by executing the subset of relevant traced tests among the whole test suite.

RQ2. What is the observed behavioral correctness level of the code co-evolution?

After tracing the tests, we could execute them to observe their effect before and after co-evolution of the code. Table 5.5 depicts the results for the original and evolved projects' versions. The second line gives the number of traced class tests and the rest of the lines categorizes the tests. We overall observe no significant difference between projects with automatically generated tests and projects with manually tests.

The most interesting project is the first $[P1_{V1}]$ to $[P1_{V2}]$. Even though the tests decreased by 161 (455 - 294 from Figure 5.4), the number of passing tests decreased only by 9 (106 - 97 from Table 5.5). Regarding the error tests, they decreased by 155. However, the failing tests increased by 3 from 2 to 5, as shown in Table 5.5. There was also the appearance of one failing test in $[P8]$. These cases of increased failing tests indicate that the code co-evolution may be not completely behaviorally correct.

Moreover, in the other projects $[P2][P3][P7][P8]$, many tests that existed in the original

Table 5.5 – Selected tests before and after code co-evolution.

Projects	$[P1_{V1}]$ to $[P1_{V2}]$	$[P2_{V1}]$ to $[P2_{V2}]$	$[P3_{V1}]$ to $[P3_{V2}]$	$[P4_{V1}]$ to $[P4_{V2}]$	$[P5_{V1}]$ to $[P5_{V2}]$	$[P6_{V1}]$ to $[P6_{V2}]$	$[P7_{V1}]$ to $[P7_{V2}]$	$[P8_{V1}]$ to $[P8_{V2}]$	$[P9_{V1}]$ to $[P9_{V2}]$
N^o class tests	114 - 57	51 - 4	8 - 2	5 - 10	2 - 3	2 - 3	11 - 12	32 - 8	1 - 1
N^o pass	106 - 97	124 - 0	14 - 1	10 - 22	2 - 3	2 - 2	206 - 68	146 - 10	5 - 5
N^o fail	2 - 5	1 - 1	0 - 0	1 - 3	0 - 0	0 - 0	1 - 0	0 - 1	0 - 0
N^o error	347 - 192	25 - 5	6 - 1	2 - 27	0 - 0	3 - 3	76 - 50	27 - 6	0 - 0

version were not in the evolved version due to the delete changes of the metamodels. This is actually a sign of behavioral correct co-evolution, as indeed the tests should be removed following the removal of the generated code for those deleted metamodel elements. In the rest of the projects $[P4][P5][P6][P9]$, roughly the same number of tests in the original version behaved the same in the evolved version, suggesting again a behaviorally correct co-evolution. These results should help developers to further check the code co-evolution rather than simply accepting them in particular when it is fully automated.

***RQ*$_2$ insights:** Our traced tests could hint in two projects that the co-evolution may not be entirely correct due to more failing tests and fewer passing tests. The rest of the project would hint on rather a correct co-evolution due to delete metamodel changes. Overall, automating the help for checking of the behavioral correctness of the code co-evolution for developers, regardless of whether they tests are manually or automatically written.

RQ3. What are the observed gains (w.r.t. test case reduction and execution time) obtained from our approach of tracing impacted tests by metamodel evolution ?

With the traced tests due to the metamodel evolution, we can assess the gains in terms of test reduction and execution time compared to the whole test suite as a baseline. Indeed, rather than re-running the whole test suite both in the original and evolved versions, or worse, not considering the tests at all. We provide developers with a zoomed view of traced impacted tests by the metamodel evolution, hence, focusing on assessing the code

co-evolution.

The first line of Table 5.5 already gives the number of traced impacted test classes that are always less than the original number of test classes.

Table 5.6 further illustrates the differences between the original test suite and the traced impacted tests in terms of number of test cases and execution time. Columns 2 and 3 give the number of original tests and of traced tests. Column 4 depicts the gains in terms of test reduction percentage. On average, we observe a reduction gain of 88% of test cases, varying from 46% to 99.9%. Out of 34612 tests in the 18 projects, we traced 1608 impacted tests representing an absolute 95% reduction.

This naturally leads to a gain in terms of execution time reduction of the tests. Columns 5 and 6 give the execution times for the whole test cases and the traced ones. Herein, we measured the execution time through IDE runner for the Junit tests. Column 7 depicts the gains in execution time of the traced tests compared to the whole test suite. On average, we observe a reduction of 84%, varying from 69% to 99%. Overall, we observe no significant difference in benefit of reducing tests and the gain in execution time between projects with automatically generated tests and projects with manually tests. Respectively, we observe a reduction gain in tests of 88% versus 81% and a reduction gain in execution of 82% versus 95.5%.

***RQ*$_3$ insights:** Tracing the metamodel evolution changes up to the impacted tests allows assessing the co-evolution behavioral correctness, while gaining, on average, a reduction of 88% in the number of tests and 84% in execution time. The reduction gains are similar with no significant difference regardless of whether the tests are manually or automatically written.

5.4 Threats to Validity

This section discusses threats to validity [220].

5.4.1 Internal Validity.

To be able to trace the impact of metamodel changes to the tests, we had to have a test suite in the selected projects. However, we observed that the Eclipse projects relying on metamodels do not come with the test suite. This was not only the case of OCL[218], papyrus [219] or Modisco [204], but also other Eclipse languages [236], [237]. Therefore, we

Table 5.6 – Reduction gains of the number of traced tests and their execution time.

Projects co-evolved in response to the evolved metamodels	N^o of tests	N^o of traced tests	Reduction gain in tests	Execution time of tests (s)	Execution time of traced tests (s)	Reduction gain in execution time
$[P1_{V1}]$ ocl.examples.pivot	7322	455	↘ 94%	339.411	48.322	↘ 86%
$[P1_{V2}]$ ocl.examples.pivot	4990	294	↘ 94%	228.88	70.856	↘ 69%
$[P2_{V1}]$ ocl.examples.base	2320	150	↘ 93.5%	123.254	26.518	↘ 79%
$[P2_{V2}]$ ocl.examples.base	2133	6	↘ 99.7%	99.105	5.294	↘ 95%
$[P3_{V1}]$ ocl.pivot	8795	20	↘ 99.7%	859.69	0.497	↘ 99%
$[P3_{V2}]$ ocl.pivot	6396	2	↘ 99.9%	261.792	12.133	↘ 95%
$[P4_{V1}]$ papyrus.infra.extendedtypes	135	13	↘ 90%	16.94	2.924	↘ 83%
$[P4_{V2}]$ papyrus.infra.extendedtypes	248	52	↘ 79%	17.076	2.957	↘ 83%
$[P5_{V1}]$ papyrus.infra.extendedtypes.emf	104	2	↘ 98%	6.912	2.11	↘ 70%
$[P5_{V2}]$ papyrus.infra.extendedtypes.emf	104	3	↘ 97%	7.31	1.802	↘ 75%
$[P6_{V1}]$ papyrus.uml.tools.extendedtypes	75	5	↘ 93%	5.9	1.505	↘ 75%
$[P6_{V2}]$ papyrus.uml.tools.extendedtypes	75	5	↘ 93%	6.246	1.099	↘ 82%
$[P7_{V1}]$ org.eclipse.modisco.infra.discovery.benchmark	524	283	↘ 46%	7.332	2.04	↘ 73%
$[P7_{V2}]$ org.eclipse.modisco.infra.discovery.benchmark	619	118	↘ 81%	14.534	4.107	↘ 72%
$[P8_{V1}]$ org.eclipse.emf.test.core	322	173	↘ 46%	176.372	2.908	↘ 98%
$[P8_{V2}]$ org.eclipse.emf.test.core	322	17	↘ 94%	157.041	0.346	↘ 99%
$[P9_{V1}]$ org.eclipse.emf.test.xml	64	5	↘ 92%	3.764	0.247	↘ 93%
$[P9_{V2}]$ org.eclipse.emf.test.xml	64	5	↘ 92%	3.193	0.257	↘ 92%

were obliged to generate the test suite with a state-of-the-art available tool EvoSuite [158]. Even though, automatic test generation is a best practice, there is the risk of having tests that are different from manually written tests. However, this does not pose a risk to our approach as the main algorithm of our approach is generic and would be able to trace the impact of a metamodel change in the same way for both automatically generated tests as well as manually written tests. This is what we observed with the fourth EMF case study having manually written tests. Indeed, the overall results of tracing tests and reduction gains were not significantly different in all case studies regardless of whether the tests are automatically generated or manually written. The main risk is related to the behavioral correctness of the code co-evolution. To check it, we require unit tests that target single methods. However, automatic test generation can even be more advantageous herein. Indeed, it generates tests for all public methods, whereas developers tend to manually write only few tests for some targeted methods. Thus, if relying only on manually written tests, there is a high risk of not assessing the behavioral correctness of many cases of code co-evolutions that are not covered by test cases. This is mitigated by relying on EvoSuite that generates a full test suite of unit tests with Junit assertions. This tool has shown his efficiency in test generation as a state-of-the-art tool [221], [222]. Indeed, this is observed when computing the coverage metric for each of our considered projects (see Table 5.4). The highest coverage is on projects with automatically generated tests and the lowest coverage is on the two projects with manually written tests.

Therefore, our approach not only checks the correctness of the code co-evolution with the traced tests, but also favors the best practice of tests generation in each release of a software language after its metamodel evolution.

Finally, as our tracing approach relies on the quality of detected metamodel changes, we analyzed, in our evaluation, each detected change and checked whether it occurred between the original and evolved metamodels. This alleviates the risk of relying on an incorrect metamodel change that would degrade the tracing of the impacted tests by metamodel changes, i.e., not tracing an impacted test by a non-considered metamodel change. In addition, as we did not have the ground truth, we could not report on the precision and recall of our approach. However, our approach uses the Code Call Graph and starts from the generated code elements corresponding to the evolved metamodel elements to recursively trace back any existing tests. Thus, by design we actually detect all the impacted tests that must be traced. To test that our algorithm does not miss any impacted test, and does not trace non-impacted tests, we manually verified the ground

truth for our smallest data set projects [$P6$] and [$P9$], because they have fewer tests and are less complex. We checked for every metamodel change all the impacted tests and we found that our approach traces all of them. Further evaluation on a ground truth is left for future work.

5.4.2 External Validity.

We implemented and evaluated our approach for EMF/Ecore metamodels and Java code with Junit tests. Other languages, such as C# or C++, use a different syntax, but conceptually use the same constructions as in Java. Although we think that the tracing would be applicable for other languages, we cannot generalize our results. Further experimentation on other languages is necessary. However, the only requirement to apply our approach to other languages is to have access to the ASTs of the parsed code and tests, and to adapt our tracing of the tests in the build call graph. Moreover, our evaluation was performed on Eclipse projects from five languages. Thus, we cannot generalize our findings to all software languages or DSLs. We also cannot generalize our results on manually written tests, in particular the test verdicts of the traced tests, i.e., pass, fail, and error. Further experimentation remains necessary and is left for future work.

5.4.3 Conclusion Validity.

Our evaluation gave promising results, showing that we could trace the impact of metamodel changes till the tests, and hence, check the behavioral correctness of the code co-evolution in practice for real-world projects. However, even though we evaluated our approach on 18 projects of metamodel evolution and code co-evolution with automatically generated tests and manually written tests, further evaluation is needed on more case studies to have more insights and statistical evidence. Finally, our user study experiment suggesting our approach usefulness needs to be replicated with more participants.

5.5 Conclusion

This chapter proposes an automated tracing of the impacted tests due to metamodel evolution. Thus, by tracing the tests before and after code co-evolution, we check its behavioral correctness. Our approach takes as input the metamodel changes and then finds the different pattern usages of each metamodel element in the code. After that, we

recursively search for its usages in the code call graph until reaching the tests. Thus, we end up matching metamodel changes with impacted code methods and their corresponding tests. We further implemented our approach in an Eclipse plugin that allows to trace the tests, map them with state-of-the-art solution GumTree and execute them. We then report them back as a diagnostic to the developers for an easier in-depth analysis of the effect of metamodel evolutions rather than re-running and analyzing the whole test suite.

The user study experiment we conducted showed that tracing manually the tests impacted by the evolution of the metamodel is a hard and error-prone task. Not only the participants could not trace all tests, but they even wrongly traced non-impacted tests. We then evaluated our approach on 18 Eclipse projects from OCL, Modisco, Papyrus, and EMF over several evolved versions of metamodels. Four projects had manually written tests and we generate tests for the other 14 projects. The results show that we successfully traced the impacted tests automatically by selecting 1608 out of 34612 tests due to 473 metamodel changes.

When running the traced tests before and after co-evolution, we observed two cases indicating possibly both behaviorally incorrect and correct code co-evolution. Thus, helping the developers to locate the code co-evolution to investigate in more detail. Furthermore, our approach provided gains that represent, on average, a reduction of 88% in number of tests and 84% in execution time. No significant difference was observed between projects with manually written tests and automatically generated ones.

Chapter 6

An Empirical Study on Leveraging LLMs for Metamodels and Code Co-evolution

As I explained in Section 2.4 of the background and Section 3.5 of the state of the art, LLMs are widely explored in Model-Driven Engineering and Software Engineering, but no existing study evaluated the LLMs capabilities to support developers in the problem of metamodels and code co-evolution. In this chapter, I present a novel approach to mitigate the challenge of metamodel evolution impacts on the code using LLMs (Challenge (C3)). Our approach is based on prompt engineering, where we design and generate natural language prompts to best co-evolve the impacted code due to the metamodel evolution. I first explain how a prompt template is structured, then how I investigated the usefulness of this template structure.

Section 6.1 presents a motivating example to illustrate the problem of metamodels and code co-evolution. Section 6.2 presents our approach for generating prompts and details the prototype implementation. Section 6.3 details our followed methodology in this empirical study. This section assesses the usefulness of my approach and compares ot to IDE quick fixes as a baseline. Section 6.3.5 reports on the obtained results and discusses threats to validity. Finally, Section 6.5 concludes this chapter.

6.1 Motivating example

To illustrate the adressed challenge, let us have an example. Figure 4.1 shows an excerpt of the version 0.9.0 of "Modisco Discovery Benchmark" metamodel. Modisco is an academic initiative project implemented in the Eclipse platform that has evolved numerous times in the past to support the development of model-driven tools, reverse engineering, verification, and transformation of existing software systems [205], [206]. Figure 4.1 illustrates some of

the domain concepts **Discovery**, **Project**, and **ProjectDiscovery** used for the discovery and reverse engineering of an existing software system. From these metaclasses, a first code API is generated, containing Java interfaces and their implementation classes, a factory, a package, etc. Listing 6.1 shows a snippet of the generated Java interfaces and classes from the metamodel in Figure 4.1.

The generated code API is further enriched by the developers with additional code functionalities in the "Modisco Discovery Benchmark" project and its dependent projects as well. For instance, by implementing the methods defined in metaclasses and advanced functionalities in new classes. Listing 6.2 shows the two classes `Report` and `JavaBenchmarkDiscoverer` of the additional code (*Line* 4, 8 in the same project "Modisco Discovery Benchmark" and in another dependent project, namely the "Modisco Java Discoverer Benchmark" project). In version 0.11.0, the "Modisco Discovery Benchmark" metamodel evolved with several significant changes, among which the following impacting changes: *1)* Renaming the property `DicoveryDate` of the class `JavaBenchmarkDiscoverer` to `DiscoveryDate`, and *2)* Moving the property *discoveryTimeInSeconds* from metaclass `Discovery` to `DiscoveryIteration`.

After applying these modifications, the code of Listing 6.1 is re-generated from the evolved version of the metamodel, which impacts the existing additional code depicted in Listings 6.2.

Listing 6.1 – Excerpt of the generated code in org.eclipse.modisco.infra.discovery.benchmark.

```
1   //Discovery Interface
2   public interface Discovery extends EObject {
3       double getTotalExecutionTimeInSeconds();
4       void setTotalExecutionTimeInSeconds(double value);
5       ...
6   }
7   //Project Interface
8   public interface ProjectDiscovery extends Discovery {...}
9   //DiscoveryImpl Class
10  public class DiscoveryImpl extends EObjectImpl implements Discovery {
11      public double getTotalExecutionTimeInSeconds() {...}
12      public void setTotalExecutionTimeInSeconds(double totalExecTime) {...}
13      ...
14  }
```

Listing 6.2 – Excerpt of the additional code V1.

```
1
2   public class Report {
```

```
3         ...
4         discovery.(*\ul{setDiscoveryTimeInSeconds}*)(...);
5     }
6
7     public class JavaBenchmarkDiscoverer extends AbstractModelDiscoverer<IFile> {
8         ...
9         discovery.(*\ul{setDicoveryDate}*)(new Date());
10        ...
11    }
```

Listing 6.3 – Excerpt of the additional code V2.

```
1
2     public class Report {
3         ...
4         discovery.(*\ul{getIterations().get(0).}*)
5         (*\ul{setDiscoveryTimeInSeconds}*)(...);
6         ...
7     }
8
9     public class JavaBenchmarkDiscoverer extends AbstractModelDiscoverer<IFile> {
10        ...
11        discovery.(*\ul{setDiscoveryDate}*)(new Date());
12        ...
13    }
```

The resulting errors in the original code in version 0.9.0 are underlined in red in Listing 6.2. Listing 6.3 presents the final result of the manual developer's co-evolution in version 0.11.0. The co-evolved code is underlined in green. The changes *rename* of the property *DicoveryDate* and the *move* of the property *discoveryTimeInSeconds* impact their usages (*Line* 4, 8 in Listing 6.2). The impact of renaming *DicoveryDate* is co-evolved by replacing *setDicoveryDate* by *setDiscoveryDate*. The impact of moving the property *discoveryTimeInSeconds* is co-evolved by extending the call path of the method *setDiscoveryTimeInSeconds* through the reference *iterations* by calling the method *getIterations* and getting the first element of the returned list of DiscoveryIteration objects.

Developers unfortunately manually co-evolve the code, which is tedious, error-prone, and time-consuming. One help developers get is from the IDE and the provided quick fixes. For example, when using Eclipse quick fixes to co-evolve these errors, it suggests creating the method `setDiscoveryTimeInSeconds` in the class `Discovery`, which does not meet the required co-evolutions shown in Listing 6.3.

With the ever-growing popularity and promising results of LLMs, a developer can prompt an LLM to suggest a co-evolution. For example, we asked ChatGPT to co-evolve the error resulted from moving the property *discoveryTimeInSeconds* by giving the erroneous

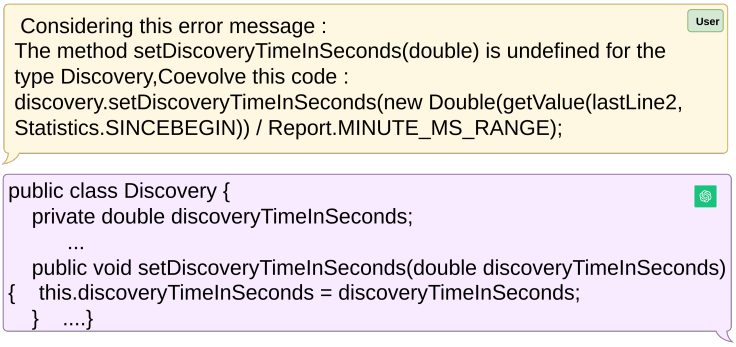


Figure 6.1 – ChatGPT primitive answer to the naive prompt.

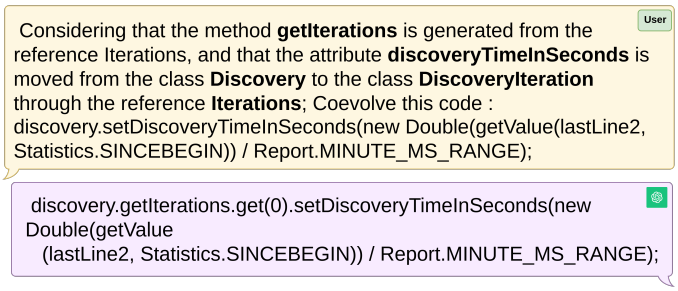


Figure 6.2 – ChatGPT improved answer with the enriched prompt with contextual information.

code (*Line* 4 in Listing 6.2) with the message of the error taken from eclipse Problems window. This is the first intuition when using ChatGPT because the developer does not know necessarily the metamodel change causing the error and finding it due to the abstraction gap is a tedious and error-prone task. Figure 6.1 shows that ChatGPT proposes to create a method named *setDiscoveryTimeInSeconds* in the class *Discovery*, which is totally wrong because it does not fit the causing change.

Our Hypothesis is that the LLM fails because our problem is more complex than simply repairing a code error. It must understand the original impacting metamodel change traced to the code error, as well as the abstraction gap between the two artefacts of metamodels and code. After improving the prompt, ChatGPT succeeded to give the right resolution as show in Figure 6.2. Our vision is that this contextual rich information must be injected in the prompt. Thus, the quality of the prompt is a key for the LLM to solve this problem of metamodels and code co-evolution.

The next section presents our contribution for a contextualized information rich prompts-based co-evolution of metamodel and code using LLMs.

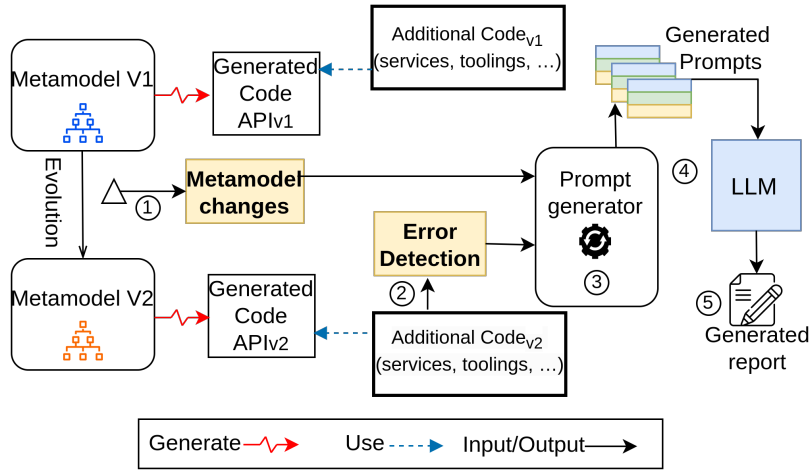


Figure 6.3 – Overall Approach for Prompt-based co-evolution.

6.2 Prompt-Based Approach

This section introduces our approach to generate the prompts needed for the code co-evolution. It first gives an overview of the approach. Then details the structure of the generated prompts, before to detail each part of it and how it is generated. Finally, it describes our prototype implementation.

6.2.1 Overview

Figure 6.3 shows the overall steps of our approach. We start by retrieving the list of changes that describe the evolution between the old metamodel and the new metamodel ①.

When the metamodel evolves, the code API is regenerated, therefore the additional code is broken. The additional code is then parsed to collect the list of errors ②. The list of changes and the list of errors are the inputs of our prompt generator. The goal is to generate a prompt for each error, including sufficient information about the change and the error itself ③. Each generated prompt is used to request ChatGPT to give a correction for the concerned error ④. A global report is generated for the additional code to allow the developer to have a visual output about the generated prompts and the answers of the LLM ⑤. Algorithm 3 further depicts the overall method of co-evolution based on a given LLM. After we parse the project, we retrieve the list of errors per class (Lines 2-3). Then for each error, we generate the prompt (Lines 4-5) and call the LLM and record its co-evolution response for analysis (Lines 6-7).

Algorithm 3: ChatGPT Co-evolution

```
   Data: EcoreModelingProject, changesList
1  javaClasses ← Parse(EcoreModelingProject)
2  for ( jc ∈ javaClasses) do
3  |  errorsList ← getErrors(jc)
4  |  for (error : errorsList) do
5  |  |  prompt ← promptGenerator(error, changesList, jc)
6  |  |  coevolutionResponse ← callLLM(prompt)
7  |  |  addToReport(error, prompt, coevolutionResponse)
8  |  end
9  end
```

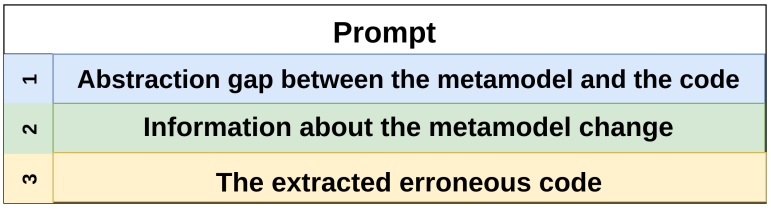


Figure 6.4 – Generated Prompt Structure.

6.2.2 Generated Prompt Structure

Figure 6.4 shows the overall structure of the envisioned prompts we will generate in order to co-evolve the code errors. In fact, the rationale behind the prompt structure is that our problem does not only concern the code errors to repair, but it is also related to the use of the code generated from the metamodels and their changes. Therefore, to contextualize our problem, we must explain : 1) what is the code generated from the metamodels, 2) what is the impacting metamodel change, and 3) what is the impacted code error to co-evolve. Concerning the prompt prefix we use "Co-evolve this code" to ask ChatGPT for one co-evolution. Next subsections detail the different parts of the prompt structure.

6.2.3 Abstraction Gap Between Metamodels and Code

One main distinction from simply repairing code errors is the interplay between the metamodels and the additional code through the generated code from the metamodels. This is due to the gap in abstraction in between metamodels and generated code. In fact, for each metamodel various code elements are generated for each metamodel element and with different patterns. Table 6.1 classifies and provides illustrative examples for

Table 6.1 – Classification of the different patterns of the generated code element from the metamodel elements.
[Examples illustrated for a metaclass *Rule*, property *Status*, and method *Execute()*]

Metamodel element type	**Generated code elements**	**Pattern of the generated code elements**	**Examples**
Metaclass	Interface	"MetaClassName"	*Rule*
	createClass() (in metamodelFactory class)	"create"+"MetaClassName"()	*createRule()*
	Literals of the class	"META_CLASS_NAME" "META_CLASS_NAME"+"_"+ "FEATURE_COUNT" "META_CLASS_NAME"+ "_"+"OPERATION_COUNT"	*RULE*, *RULE_FEATURE_COUNT*, *RULE_OPERATION_COUNT*
	Accessor of Meta objects (in metamodelPackage class)	"get"+"MetaClassName"()	*getRule()*
	Class implementation	"MetaClassNameImpl"	*RuleImpl*
	Adapter	"create"+"MetaClassName"+"Adapter"	*createRuleAdapter()*
Attribute (Same for a Reference)	Signature of getters and setters	"get"+"AttributeName"(), "set"+"AttributeName"()	*getStatus()*, *setStatus()*
	Accessor of Meta objects	"get"+"MetaClassName"+"_"+"AttributeName"()	*getRule_Status()*
	Literal	"META_CLASS_NAME"+"___"+"ATTRIBUTE_NAME"	*RULE___STATUS*
	Implementation of getters and setters	"get"+"AttributeName"(), "set"+"AttributeName"()	getStatus(), setStatus()
Method	Declaration of the method	"methodName"()	*Execute()*
	Accessor of meta objects	"get"+"MetaClass"+"___"+"MethodName"()	*getRule___Execute()*
	Literal	"META_CLASS_NAME"+"___"+"METHOD_NAME"	*RULE____EXECUTE*
	Implementation of the method	"methodName"()	*Execute()*

the different generated code elements from each metamodel element, namely metaclass, attribute/reference, and method. For example, take the case of a metaclass [1]. EMF generates a corresponding interface and a class implementation, a *createClass()* method in the factory class, three literals (*i.e.,* constants) for the class and an accessor method in the package class, and a corresponding to create adapter method. For the attribute case, EMF generates the signature and implementation of a getter and a setter, an accessor, and a literal. This classification is essential to match the different errors with the right used generated code elements to explicit the abstraction gap. If a class is changed, all its generated elements are impacted and their usages will be erroneous. For example, if a class is renamed, every invocation of the method "get+className" will be erroneous and must be co-evolved. Thus, we consider the abstraction gap as the first contextual information we inject in the prompts for the LLM to co-evolve the code errors.

1. For simplicity, we refer to a metaclass by simply a class in the remaining of the chapter.

6.2.4 Metamodel Evolution Changes

The metamodel changes that we consider as the second contextual information, are injected in the prompts for the LLM to co-evolve the code errors.

As a reminder, we use in this thesis a connection layer to our prompt generation approach. This layer represents an interface specification of changes [1] bridging our prompt generator with the existing change detection approaches [59], [61]–[65]. Accordingly, we detail the changes that we consider in Table 5.1.

6.2.5 Extracted Code Errors

Now that we have two main ingredients needed for the generation of the prompts. We only require the erroneous code to be co-evolved.

To do so, we parse the code (i.e., *compilation units*) to access the Abstract Syntax Trees (ASTs) and retrieve the code errors. Each error contains the necessary information to locate the exact impacted AST node in the parsed global AST (*i.e.,* char start and end) and to process it (*i.e.,* message). After that, we simply extract the sub-AST corresponding to the code containing the error. We consider three possible situations, namely 1) if the error is in a method, we extract the whole method, 2) if the error is in the imports, we extract the list of imports, 3) if the error is in the class definition or the fields, we extract it without the class's methods. This constitutes the final part of the contextual information we inject in the prompts for the LLM to co-evolve the code errors. Note that we simply specify in the prompt before the code the order to *"Co-evolve this code: "*.

6.2.6 Prompt Generation

Algorithm 4 allows generating prompts following the specified structure in Figure 6.4. It first finds the ASTNode corresponding to the error in the code (Line 1). Then, it iterates over the list of metamodel changes to match the error node with the code usage to identify the impacted abstraction gap (Lines 6-8). After that, it summarizes the impacting metamodel change (Line 11). Finally, it extracts the erroneous code (Line 21) and puts together all three contextual information into one generated prompt (Lines 22-25). Figure 6.5 shows an example of the generated prompt for the error in Listing 6.2 (Line 4) due to the move of property *DiscoveryTimeInSeconds* in the metamodel.

	Prompt
1	The method **setDiscoveryTimeInSeconds** is generated from The attribute **discoveryTimeInSeconds**
2	The attribute **discoveryTimeInSeconds** is moved from the class **Discovery** to the class **DiscoveryIteration** through the reference **Iterations**
3	Co-evolve this code : discovery.**setDiscoveryTimeInSeconds**(new Double(getValue(lastLine2, Statistics.SINCEBEGIN)) / Report.MINUTE_MS_RANGE);

Figure 6.5 – Move attribute prompt example.

6.2.7 Prototype Implementation

We implemented our solution as an eclipse Java plugin handling Ecore/EMF metamodels and their Java code. To retrieve the list of errors, we used JDT eclipse plugin [2]. To launch our tool, we added a command to the context menu when selecting a java project. Generated prompts are sent to ChatGPT (see next section 6.3.1), more specifically, its OpenAI API endpoint "https://api.openai.com/v1/chat/completions". We used Java JSON package to send our prompts and to receive ChatGPT responses. The error location, the corresponding generated prompt and ChatGPT responses are parsed in CSV file to have a visible results and to keep the history of proposed the co-evolutions. Moreover, quick fixes (our baseline) are called using *org.eclipse.jdt.ui.text.java.IQuickAssistProcessor* and *org.eclipse.jdt.ui.text.java.IJavaCompletionProposal*.

6.3 Methodology

This section describes our methodology for empirically assessing the capabilities of ChatGPT in addressing the problem of metamodel and code co-evolution. It first describes the selected LLM and the followed evaluation process. Then, it presents our research questions and the data set.

6.3.1 Selected LLM

We chose to use ChatGPT *GPT-3.5-turbo* in chat mode. It currently points to *gpt-3.5-turbo-0613* released in June 2023. We opted for this model because of four factors. The first one is that prompt content is only textual, we don't need to inject audio or image content. The second one is its capacity to generate good answers for requests about code

2. Eclipse Java development tools (JDT): `https://www.eclipse.org/jdt/core/`

Algorithm 4: Prompt Generator Algorithm

```
   Data: error, changesList, javaClass
 1 errorNode ← findErrorAstNode(javaClass, error)
 2 found ← false
 3 while (change ∈ changesList & ¬found) do
 4 |   switch change do
 5 |   |   case RenameClass do
 6 |   |   |   if (errornode.name="get"+change.name ) then
 7 |   |   |   |   found ← true
 8 |   |   |   |   abstractionGap="The method "+ errornode.name+" is generated from
   |   |   |   |    the metaclass "+ change.name
 9 |   |   |   else if ... then
10 |   |   |   |   /*treat other abstraction gaps*/
11 |   |   |   changeInfo= "The metaclass "+ change.oldName+" is renamed to
   |   |   |    "+change.newName
12 |   |   end
13 |   |   case RenameProperty do
14 |   |   |   ...
15 |   |   end
16 |   |   case DeleteProperty do
17 |   |   |   ...
18 |   |   end
19 |   end
20 end
21 codeError ← extractCodeError(errornode)
22 prompt.add(abstractionGap)
23 prompt.add(changeInfo)
24 prompt.add(codeError)
25 return prompt
```

and models generation [15], [16], [19], [20], [192]–[194]. The third factor is that it was the latest API version that is accessible, *gpt-4* API still not accessible for us. The last one is related to the high popularity of ChatGPT as a tool. It has more than 100 million users, and its website saw more than 1.7 billion visitors in the last three months, with Software and Software Development as visitors' top category [3].

6.3.2 Evaluation Process

First, as we aim to query ChatGPT to co-evolve the erroneous code due to metamodel evolution, we need to provoke the errors in the code. To do so, we replace the original

3. `https://www.similarweb.com/fr/website/chat.openai.com/#demographics`

metamodel by the evolved metamodel. Then, we regenerate the code API with EMF. This will cause errors in the additional code that must co-evolve. After that, we must map the errors with the causing metamodel change before to generate a prompt with all appropriate information to be able to co-evolve the code errors. We then rely on the OpenAI API to query ChatGPT before to analyze its results. Finally, we measure the correctness of ChatGPT co-evolution by comparing its co-evolution with the manually co-evolved version by developers. Note that the comparison is processed manually by authors. This allows us to measure the *correctness* reached by ChatGPT. Correctness varies from 0 to 1, i.e., 0% to 100% and is defined as follows:

$$Correctness = \frac{LLMCoevolutions \cap ManualCoevolutions}{ManualCoevolutions}$$

We chose the structure shown in Figure 6.4 since it contains the contextual information needed for our problem of metamodels and code co-evolution. This structure was built after few manual naive attempts with ChatGPT, starting from a minimum context (as shown in the Motivating Example of Figure 6.1 that failed) and by enriching the structure with more context information. However, we do not claim its completeness in terms of needed information or if it is the best structure. Other variants can lead to different results. To investigate more this choice, we generated three more variations of this structure to observe its effect on the results. Table 6.2 contains each variation and its corresponding explanation. Since our prompt contains three parts, we can change the order of these parts (Order Change operator) and the size of the erroneous code we put in the prompt (Minimal Code operator). Finally, rather than asking for a single co-evolution solution, we ask for alternative ones (Alternative Answers operator) in the prompt prefix. To stress out our evaluation, we conducted 5 runs separated by almost a day, the time we needed for one run to check manually the results of each project, which means that each generated prompt is proposed to ChatGPT five times. This aims to check whether ChatGPTgives a same or different answer, hence, assess its robustness. Finally, we compare ChatGPTto a baseline, namely the IDE quick fixes that are provided to repair the code errors. Note that we do not take as a baseline ChatGPTwith prompts that only contain the code error, since as shown in Section 6.1 and Figure 6.1, it does not work.

6.3.3 Research Questions

To assess the capabilities of ChatGPTin the co-evolution of code, we set the following research questions.

Table 6.2 – Variation operators of our original prompt (OP).

Variation Operators	Explanation
Order Change (OC)	We change the order between the three structured parts of the prompt. We start by describing the metamodel change before the abstraction gap.
Minimal Code (MC)	Instead of giving the whole method that contains the code error to co-evolve, we only give the instruction of code error.
Alternative Answers (AA)	Instead of asking the LLM to give one solution of co-evolution, we specifically ask for alternative ways to co-evolve the code error.

Table 6.3 – Details of the metamodels and their evolutions.

Case study	Evolved metamodels	Versions	Atomic changes in the metamodel	Complex changes in the metamodel
OCL	Pivot.ecore in project ocl.examples.pivot	3.2.2 to 3.4.4	Deletes: 2 classes, 16 properties, 6 super types Renames: 1 class, 5 properties Property changes: 4 types; 2 multiplicities Adds: 25 classes, 121 properties, 36 super types	1 pull property 2 push properties
Modisco	Benchmark.ecore in project modisco.infra.discovery.benchmark	0.9.0 to 0.13.0	Deletes: 6 classes, 19 properties, 5 super types Renames: 5 properties Adds: 7 classes, 24 properties, 4 super types	4 moves property 6 pull property 1 extract class 1 extract super class
Papyrus	ExtendedTypes.ecore in project papyrus.infra.extendedtypes	0.9.0 to 1.1.0	Deletes: 10 properties, 2 super types Renames: 3 classes, 2 properies Adds: 8 classes, 9 properties, 8 super types	2 pull property 1 push property 1 extract super class

RQ1. *To what extent can ChatGPT co-evolve the code with evolved metamodels?* This aims to assess the ability of ChatGPT to give correct resolutions to co-evolve the code according to the metamodel changes.

RQ2. *How does varying the temperature hyperparameter affect the output of the co-evolution?* The temperature hyperparameter controls the creativity of the language model. This question aims to assess the capability of ChatGPT to co-evolve the erroneous code when given less or more creativity in the generation of the solution.

RQ3. *How does varying the prompt structure affect the output of the co-evolution?* This aims to assess the quality improvement of the co-evolutions due to the prompts' variations.

RQ4. *How does ChatGPT proposed co-evolution compare to the quick fix baseline?* As quick fixes are provided by default in an IDE to repair the code errors, this question aims to assess which method outperforms the other in the task of code co-evolution with evolving metamodels.

Table 6.4 – Details of the projects and their caused errors by the metamodels' evolution.

Evolved metamodels	Projects to co-evolve in response to the evolved metamodels	N^o of packages	N^o of classes	N^o of LOC	N^o of Impacted classes	N^o of tot errors
OCL Pivot.ecore	[$P1$] ocl.examples.xtext.base	12	181	17599	10	29
Modisco Benchmark.ecore	[$P2$] modisco.infra.discovery.benchmark	3	28	2333	1	6
	[$P3$] gmt.modisco.java.discoverer.benchmark	8	21	1947	4	30
	[$P4$] modisco.java.discoverer.benchmark	10	28	2794	9	56
	[$P5$] modisco.java.discoverer.benchmark.javaBenchmark	3	16	1654	9	73
Papyrus ExtendedTypes.ecore	[$P6$] papyrus.infra.extendedtypes	8	37	2057	8	44
	[$P7$] papyrus.uml.tools.extendedtypes	7	15	725	7	28

6.3.4 Data set

This section presents the used data set in our empirical study, to be found in the attached supplementary material [4].

We chose Eclipse Modeling Framework (EMF) platform as technological space, which allows us to build modeling tools and applications based on Ecore metamodels [223]. First, we aimed at selecting metamodels with meaningful real evolutions that do not consist in only deleting metamodel elements, but rather including complex evolution changes (see subsection 4.2.2).

This selection criterion resulted in seven Java projects from three case studies of three different language implementations in Eclipse, namely OCL [218], Papyrus [219], and Modisco [204] with their versions that was manually co-evolved by developers, that represent our ground of truth. OCL is a standard language defined by the Object Management Group (OMG) to specify First-order logic constraints. Papyrus is an industrial project led by CEA [5] to support model-based simulation, formal testing, safety analysis, etc. Modisco is an academic initiative to support development of model-driven tools, reverse engineering, verification, and transformation of existing software systems. Thus, the case studies cover standard, industrial, and academic languages that have evolved several times for more than 10 years of continuous development period.

Table 6.3 presents the details on the selected case studies, in particular about their metamodels and the occurred changes during evolution. The total of applied metamodel changes was 330 atomic changes, including 19 complex changes in the three metamodels. Table 6.4 presents the details on the size of the seven projects and code of the original versions that we co-evolve in addition to the number of errors after the metamodels

4. `https://figshare.com/s/bf35039892799c0e6f34`
5. `http://www-list.cea.fr/en/`

Table 6.5 – Measured correctness rate per temperature.

Temperature	Projects						
	P1	P2	P3	P4	P5	P6	P7
0.0	76%	62.5%	100%	80%	86%	93.7%	92.8%
0.2	**84%**	**75%**	**100%**	**82%**	**88%**	**95.8%**	**96.4%**
0.5	57%	50%	100%	58%	68%	87.5%	89.2%
0.8	50%	37%	0%	32%	46%	62.5%	85.7%
1.0	30%	29%	0%	26%	40%	68.7%	78.5%

evolution.

6.3.5 Results

This section now presents our results answering each RQ.

RQ1: Assessing the ability of ChatGPT to give correct code co-evolutions

To assess the ability of ChatGPT to give correct code co-evolutions, we ran over the errors caused by the metamodel changes and generated 266 prompts that we submitted to ChatGPT. Note that we ran our original prompts asking for a single co-evolution of the code error. We further set a fixed temperature hyperparameter to 0.2. A value of 0 restricts the generation of a solution towards a more deterministic one and the more it is higher the more creative the model gets in generating a different solution. Thus, we chose 0.2 to allow only little creativity while restricting the model to not get a different answer for the same prompts in different executions, since we ask for a single solution. Among the 266 generated prompts, ChatGPT gave, on average, a correct code co-evolution in 88.7% of the time, varying from 75% to 100%. When taking only on complex changes into consideration, correctness improves on average from 88.7% to 95.2%, *i.e.,* ChatGPT performs better for complex changes. Results show a promising ability of ChatGPT to actually co-evolve code with evolving metamodels when given the right contextual information in the prompts, rather than simply the errors and their messages.

***RQ*$_1$ insights:** Results confirm that ChatGPT is able at 88.7% to correctly co-evolve code due to metamodel evolution thanks to the information on the abstraction gap and the impacting metamodel change. It also performs better on complex changes.

RQ2: Studying the impact of temperature variation on the output co-evolutions

The temperature hyperparameter controls the creativity of the language model. The temperature hyperparameter in ChatGPT can be set between 0 and 2. Yet, it is suggested to only set it between 0 and 1 in the documentation [6]. Thus, to assess the effect of the temperature in co-evolving the code, we will vary it on 0, 0.2, 0.5, 0.8, and 1.0. Table 6.5 shows the obtained results.

Overall, we observe that as the temperature increases, the correctness of code co-evolutions suggested by ChatGPT decreases. Notably, the correctness improves when the temperature is increased from 0 to 0.2. However, it subsequently degrades with the further increase in temperature, up to 1. At temperatures 0.8 and 1, we observed the worst decrease in correctness. The best performance of ChatGPT is obtained at the temperature 0.2. Note that even with a temperature set to 0, which implies no creativity, ChatGPT yields results that are nearly as satisfactory as those obtained with a temperature of 0.2.

Moreover, when we repeated the same prompt for each temerature for 5 times, the five answers of ChatGPT were similar. ChatGPT gives almost the same proposition of co-evolution, sometimes it uses different terms to comment its answer. For example: *"// Remove the following line since DiscoveredProject is removed"* and *" // Code using DiscoveredProject should be removed"*. However, they are the same co-evolutions. Sometimes ChatGPT also uses intermediate variables to give the same co-evolution, for example: *"return aReport.generate()"* and *benchmarkModel=aReport.generate() return benchmarkModel"*. We believe that this result of obtaining the same co-evolutions over 5 runs shows the efficiency of the prompt template in Figure 6.4. By including the necessary information (i.e., abstraction gap, causing change information, and code error), it allows ChatGPT to narrow the scope of possibilities and to propose similar answer for each unique prompt in each run.

***RQ*$_2$ insights:** Results show that lower temperature of 0 and 0.2 give better co-evolutions from ChatGPT with 0.2 being the best we observed. Interestingly, we obtained the same results over 5 different runs, which suggests the efficiency our prompt structure in narrowing the scope of possibilities of ChatGPT's answers.

6. `https://platform.openai.com/docs/api-reference/chat`

Table 6.6 – Measured correctness rate for different prompt variations. [↗ and ↘ are increase and decrease in correctness.]

Variations	Projects						
	P1	P2	P3	P4	P5	P6	P7
Original Prompt (OP)	**84%**	**75%**	**100%**	**82%**	**88%**	**95.8%**	**96.4%**
Order change (OC)	84%	66%↘	100%	74%↘	86%↘	95.8%	96.4%
Minimal Code (MC)	88.4%↗	87.5%↗	100%	86%↗	96%↗	97.9%↗	96.4%
Alternative Answers (AA)	84%	79%↗	100%	92%↗	92%↗	95.8%	96.4%

RQ3: Studying the impact of prompt structure variation on the output co-evolutions

In our approach, we propose a possible structure for the generated prompts (cf. Subsection 6.2.6). However, there is no assurance that this represents the best proposition. Therefore, we varied the structure that we proposed in three different ways, as shown in Table 6.2. Then we ran the three variations of the original prompts in order to assess the fluctuation and effect on the correctness of the proposed co-evolutions. Here we set the temperature to 0.2 based on results of RQ2.

Table 6.6 shows the obtained results. Overall, we observe slight increase and decrease compared to the original Prompts structure (**OP**). We observe that changing the order in the prompt by first describing the metamodel change (**OC**) decreases the correctness of the proposed ChatGPT co-evolutions. It decreased by -9% in P2, -8% in P4 and by -2% in P5. We observed that in particular when describing the abstraction gap with the generated elements for the metamodel deleted classes, ChatGPT assumes that the code classes still exist in the code and were not deleted. Thereby, altering sometimes the correctness of its proposed co-evolutions.

However, the two other prompt variations of giving a minimal code containing the error and asking for alternative solutions delivered better overall results. Surprisingly, on the one hand, giving only the code instruction containing the error (**MC**) gave the best results. It improved by +4.4% in P1, +12.5% in P2, +4% in P4, +8% in P5, and +2.1% in P6. Our observation is that this improvement is sometimes due to the inability of ChatGPT to find the impacted code element and co-evolve it within complex and long methods. On the other hand, when asking for alternatives, it improved only by +4% in P2, +10% in P4, and +4% in P5. We observed that when ChatGPT is unable to find the correct resolution

with (**OP**) prompts, it is unlikely to find the right one when asking for alternatives (**AA**). Only in few cases it could find the correct co-evolutions. The generation of prompts and saving the results took on average from about 10 seconds per prompt for the Original Prompts to 84 seconds per prompt in the case of Alternative Answers (AA) variation prompts. Finally, when focusing only on complex changes, correctness improves on average from 88.7% to 97.6% for **(MC)** and **(AA)**. This implies that, overall, ChatGPT also performs better for complex changes when varying the prompts. This can be explained by the fact that complex changes provide much more context information that guide better ChatGPT to give better responses.

***RQ*$_3$ insights:** Results show an improvement with two variants out of the three we explored. However, gains are not significant compared to our original structure of the prompt. Variants also perform better on complex changes.

RQ4: Comparison with quick fixes as baseline

To compare to a baseline the obtained results of proposed co-evolutions with our generated prompts, we checked what is the best quick fix an IDE proposes for each error. Algorithm 5 shows the followed steps to run the quick fixes on our projects automatically. It iterates over the java classes and for each error, we automatically apply the top quick fix suggested by the IDE. It stops when all the errors disappear or if the remaining errors have no possible quick fixes purposed by the IDE.

In Table 6.7, we present the percentage of errors that quick fixes eliminated for each project. While the quick fixes eliminated from 41% to 100% errors. We use the term eliminated instead of correct co-evolution because no quick fix was applied as expected to the manual developers' co-evolutions. In other words, the correctness rate of automatic quick fixes are equal to 0 and are not suited for the task of metamodels and code co-evolution. For example, concerning errors caused by class or property deletion from the metamodel, renaming a class or an attribute, moving, pushing or pulling attributes or methods from a class to another, the quick fixes proposed to create them back in their old containers. This is in contradiction of the applied metamodel changes and the code co-evolution.

For errors caused by changing a variable's type, the quick fixes always proposed to add a cast with a wrong type.

Similarly, as naive prompts with only the code errors and their messages, quick fixes

Algorithm 5: Quick fixes for coevolution

```
   Data: EcoreModelingProject
 1 javaClasses ← Parse(EcoreModelingProject)
 2 for ( jc ∈ javaClasses) do
 3 |   errorsList ← getErrors(jc)
 4 |   while (!errorsList.isEmpty() & hasQuickFix) do
 5 |   |   error ← errorsList.next()
 6 |   |   if error.hasQuickickFix() then
 7 |   |   useQuickFixes(error)
 8 |   |   refreshJavaClass(jc)
 9 |   |   refreshErrorsList(jc, errorsList)
10 |   end
11 end
```

do not take in consideration the knowledge of the abstraction gap and the information contained in its causing metamodel changes. For example, the quick fix `create the missing method m()` is applied no matter the metamodel change (i.e., deletion, moving, pulling, or pushing changes) and no matter the metamodel element it was generated from. Our approach of generated prompts takes into account the context of the impacted code, the abstraction gap, and the causing metamodel change thanks to the prompt template that we designed (cf. Table 6.4).

***RQ*$_4$ insights:** Results show that ChatGPT with our generated prompts completely outperforms the quick fixes in correctly co-evolving the code.

6.4 Threats to Validity

This section discusses threats to validity w.r.t. Wohlin et al. [220].

6.4.1 Internal Validity.

A first internal threat is that we only varied the temperature hyperparameter over ChatGPT API. The documentation suggests not to modify top_p and temperature at the same time, so we chose to let the default value of top_p = 1. Moreover, to measure the correctness, we analyzed the developers' manual co-evolution. To mitigate the risk of misidentifying a manual co-evolution, for each impacted code error, we mapped it in

Table 6.7 – Number of applied Quick Fixes for each project and per evolved metamodel.

Evolved metamodels	Co-evolved projects	% of eliminated errors
OCL Pivot.ecore	[$P1$]	41%
Modisco Benchmark.ecore	[$P2$]	100%
	[$P3$]	100%
	[$P4$]	83%
	[$P5$]	67%
Papyrus ExtendedTypes.ecore	[$P6$]	69%
	[$P7$]	93%

the co-evolved class version. If we did not find it, we further searched in other classes in case the original impacted part of the code was moved into another class. Thus, our objective was to reduce the risk of missing any mappings between an error in the original code and its co-evolved version. Moreover, as our co-evolution relies on the quality of detected metamodel changes. We also validated each detected change by checking whether it occurred between the original and evolved metamodels. This alleviates the risk of relying on an incorrect metamodel change that would degrade the generated prompts and lead to wrong co-evolution from ChatGPT.

6.4.2 External Validity.

We implemented and ran the empirical study for EMF and Ecore metamodels and Java code. Note that the choice of Java is imposed by EMF and no other languages are supported. Thus, we naturally do not generalize to other languages. We also relied only on ChatGPT and GPT-3.5-turbo released in june 2023. Therefore, we cannot generalize our approach to other LLMs and other future versions of ChatGPT. It is also unclear how our findings would transfer to other benchmarks other than EMF Ecore metamodels and java code. Further experiments are necessary in future to get more insights and- before any generalization claims.

6.4.3 Conclusion Validity.

Our empirical study show promising results with ChatGPT being able to generate correct code co-evolution when given the necessary contextual information. It showed to be useful, with an average of 88.7% correctness (from 75% to 100%). The results also show the benefit over relying on quick fixes. Nonetheless, even though we evaluated it on real evolved projects, we must evaluate it on more case studies to have more insights and statistical evidence.

6.4.4 Discussion and Limitations

The rationale behind the prompt structure, which an important part in our empirical study, is that our problem concerns the use of the code generated from the metamodels and their changes and not only error repair. Moreover, our results show that ChatGPT can co-evolve the code correctly by setting lower temperature, especially in case of complex metamodel changes. Furthermore, repeating the experimentation five time has led to the same results shown in Table 6.5 and Table 6.6, which confirms the robustness of ChatGPT in the code co-evolution task and that his capability is not due to randomness. Finally, we handled a single metamodel with changes that are independent between them. Treating the case of multiple metamodel and the case of interdependent changes would need setting an order of priority between them, which we left for a future work.

6.5 Conclusion

Co-evolving the impact of metamodel evolution on the code is costly and yet challenging. In this chapter, we propose a prompt-based approach for metamodel and code co-evolution. This approach relies on designing and generating a rich contextual information that we inject in the prompts, namely the abstraction gap knowledge, the metamodel changes information and the impacted code to be co-evolved. We evaluated our approach on seven EMF projects and three evolved metamodels from three different Eclipse EMF implementations of OCL, Modisco and Papyrus, with a total of 5320 generated prompts. Results show that on average ChatGPT has successfully proposed 88.7% of correct co-evolutions with our original generated prompts. We evaluated then the impact of the temperature variation on the proposed co-evolutions. We found that ChatGPT gives better responses with lower temperature values of 0 and 0.2. Moreover, when experimenting other

variations of the structure of generated prompts, we observed that there was improvement in two variations. The first one is giving the minimum impacted code to co-evolve in the prompt, and the second one is requiring alternative answers for the co-evolution. However, varying the prompt by changing the order of the contextual information degraded a little the quality of the proposed co-evolutions. Finally, we compared our approach with the quick fixes of the IDE as a baseline. Results show that our approach significantly outperforms the quick fixes that do not take into account the context of the abstraction gap and the metamodel changes.

Chapter 7

Conclusion

This chapter summarizes the contributions (Chapter 4, 5 and 6) of this dissertation.Each contribution is briefly revisited to highlight its results in addressing the research objectives and its limitations. Additionally, this chapter explores potential future directions and perspectives for each contribution.

7.1 Summary of contributions and limitations

In this thesis, I started by proposing an automatic code co-evolution approach with evolving metamodel. This approach effectively automates code co-evolution, resolving 100% of errors, that we can divide into direct and indirect errors, and reaching 82% of precision and 81% of recall. The automation of the approach allows significantly reducing manual effort. The strategy prioritizes minimal deletions when possible, ensuring that necessary changes do not remove more code than required to apply the metamodel changes. Moreover, test execution results showed no significant behavioral differences between the original and co-evolved code, reinforcing the correctness of the automated approach in preserving code functionality. Additionally, the method is computationally efficient, with an average resolution time of less than half a second per error, making it scalable even on standard hardware. The comparison with IDE quick fixes highlights their limitations in handling co-evolution scenarios. Quick fixes omit contextually information of metamodel changes, leading to applying different fixes than the expected developers' resolutions. Finally, the comparison with the semi-automatic co-evolution approach [11] highlights a key trade-off between automation and precision with a difference of almost -10%, knowing that the automatic approach logs the detailed co-evolution operations in a report that developers can check later.

While our approach demonstrates strong performance in automating code co-evolution, it has several limitations. First, it does not handle cases where multiple, interdependent metamodels evolve simultaneously, which can introduce additional complexity. Additionally,

the evaluation is constrained by the size of the projects and the number of resulting compilation errors, limiting the generalizability of the results. Another limitation is the sole focus on EMF dataset, without an assessment its applicability in other technological spaces. Furthermore, the evaluation does not cover all possible metamodel changes and error scenarios, meaning certain edge cases may not be addressed. Lastly, our testing relies only on EvoSuite-generated tests, without incorporating manually written tests, that we argued its usefulness, but may not be complete in the context of out evaluation.

Second, I developed an approach for automated testing of code co-evolution. Before tackling the evaluation of the approach itself, I conducted a user study to highlight the challenges of manually tracing impacted tests after metamodel evolution which can be both incomplete and error-prone. Since the approach automated testing of code co-evolution is based on tracing the impacted set of tests before and after co-evolution, I had to confirm that the approach successfully traces impacted tests due to metamodel changes, isolating the relevant tests. At this point of the evaluation, I included projects with both manually written and automatically generated tests. The correctness of the co-evolution is indicated by the fluctuation in the number of passing, failing, or erroneous tests. Overall, I found that running the traced subset of tests can indicate if the co-evolution was processed correctly or not, regardless of whether they tests are manually or automatically written, thus gaining effort and time.

While our approach shows promising results, it has several limitations. The size of the projects and the number of both manually written and generated tests present challenges, as larger projects with extensive test suites may require additional optimization for efficient handling. Additionally, our evaluation was conducted only on EMF dataset, limiting its applicability to other technological spaces. Future work should explore its effectiveness in different contexts. Moreover, the visual output of our tool remains basic and needs improvement to better assist developers in the code co-evolution process by providing clearer insights and more intuitive interactions.

Finally, with the emergence of Large Language Models, I Investigated in my empirical study the ability of Chatgpt to give correct co-evolutions. Setting the hyperparameter temperature to 0.2, I obtained 88.7% of correctness rate in handling code co-evolution, improving to 95.2% for complex metamodel changes, thus highlighting ChatGPT's promising potential. Then, I analyzed the impact of varying the temperature on the results. I found that lower temperature of 0 and 0.2 give better coevolutions. Interestingly, we obtained the same results over 5 different runs, which suggests the efficiency our prompt structure

in narrowing the scope of possibilities of ChatGPT's answers. Furthermore, I studied the impact of varying the prompt structure of the prompts on the quality of Chatgpt proposed co-evolutions. Providing only the code instruction containing the error yielded the best results, in some cases, as it helped ChatGPT focus on the specific issue. Asking for alternative answers also improved accuracy but to a lesser extent. However, changing the order by describing the metamodel change first instead of the describing the abstraction gap between the metamodel impacted element and the erroneous code led to a drop in correctness, as ChatGPT sometimes assumed deleted elements still existed. Thus, showing that well-structured prompts significantly enhance ChatGPT's performance. Finally, the comparison of the proposed co-evolutions with quick fixes revealed significant limitations in the latter. The lack of context-awareness in IDE quick fixes and understanding of the abstraction gap made them unsuitable for the co-evolution task. In contrast, our prompt generation approach addresses these gaps by considering the impacted code's context and the metamodel changes, guided Chatgpt to propose more accurate co-evolutions.

Regarding limitations of this study include the fact that the empirical evaluation was conducted only on GPT-3.5, without testing other Large Language Models, which could provide greater confidence in the results. Additionally, among the LLM hyperparameters, only the temperature was varied while top_p remained fixed. A more comprehensive analysis involving the variation of all hyperparameters could offer deeper insights into the model's capability in proposing co-evolutions. Furthermore, the study was based on a prototype implementation, and there is not yet a fully developed tool to assist developers in code co-evolution based on our approach.

7.2 Perspectives

7.2.1 Automated co-evolution of metamodels and code

We first plan to explore ordering the errors in the code before co-evolving them. In fact, we co-evolve the errors in the same order they are retrieved by the Java Develomplent Tools (JDT). Thus, we will investigate whether it could reach better correctness with faster co-evolution or not.

We also plan to evaluate our approach on more case studies and on the case of simultaneous multiple metamodel evolution. Finally, we plan to extend our resolutions with a replace resolution, based on distance metrics to find element replacements to co-

evolve the code. After that, we can rely on our approach to extend it with a search-based heuristics, such as genetic algorithms to explore different paths of co-evolution, in contrast to a single path that we compute in this paper. In particular, to explore the different alternative resolutions of $CR1 - CR4$.

7.2.2 Automated testing of metamodels and code co-evolution

First, we plan to improve the performance of our implementation with optimization of the tests' tracing. We also plan to extend our approach to projects that use an equivalent form of metamodels in other technological space than Eclipse, such as JHipster and OpenAPI that both generate code from a model specification similar to a metamodel. Thus, we can have alternative case studies.

After that, we plan to investigate the techniques of test amplification on the selected tests we traced from the metamodel changes. Indeed, once we select a subset of tests, we could amplify them by generating more similar tests, yet, with different assertions to cover more corner cases. This would amplify the behavioral check of the code co-evolution.

Moreover, the knowledge about the generated code elements from the metamodel elements (e.g., getter/setter for EAttribute, class/interface for EClass, etc.) is so far hard-coded in the implementation. The mappings must be provided for our approach to be able to trace the tests. We could express these mappings in a generic way as a configuration file taken as input in our approach. Thus, we could reuse our approach more easily in other scenarios than EMF Eclipse (e.g., jHipster), given their corresponding mappings as input. This is left as future work to generalize our approach.

Finally, we will also explore another type of amplification, which is the interchange of the tests between the original and evolved versions. In other words, we aim to co-evolve the tests of the original and evolved versions, respectively forward and backward to the evolved and original versions, while removing duplicates.

7.2.3 An Empirical Study on Leveraging LLMs for Metamodels and Code Co-evolution

First, we intend to evaluate or approach in other technological spaces than EMF, such as OpenAPI and the challenge of the API evolution impact on clients' code. After that, we plan to transform our approach into a DSL-based approach with a graphical user interface for the output report instead of a CSV file. To facilitate prompt generation and enhance

the option of prompt variation, a DSL would be a viable solution. We also plan to replicate our empirical study with other LLMs and other contexts of co-evolution (e.g., between code and test [230]). Another actionable element is to investigate the mining of contextual information from Software Engineering tasks to enrich the prompts, and then improve baseline results.

Finally, since our contributions focus on empirically studying the use of ChatGPT in metamodel and code co-evolution, we plan to implement an alternative of the quick fix engine in the Eclipse IDE based on our generated prompt structure. Integrating our prompt-based metamodel and code co-evolution in the IDE will have a direct impact in helping MDE developers and language engineers.

BIBLIOGRAPHY

[1] Y.-T. Chen, C.-Y. Huang, and T.-H. Yang, « Using multi-pattern clustering methods to improve software maintenance quality », *IET Software*, vol. 17, *1*, pp. 1–22, 2023. DOI: https://doi.org/10.1049/sfw2.12075. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/sfw2.12075. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/sfw2.12075.

[2] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[3] J. Cabot and M. Gogolla, « Object constraint language (ocl): a definitive guide », *in Formal methods for model-driven engineering*, Springer, 2012, pp. 58–90.

[4] M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed, « Towards model-and-code consistency checking », *in 2014 IEEE 38th Annual Computer Software and Applications Conference*, IEEE, 2014, pp. 85–90.

[5] G. Kanakis, D. E. Khelladi, S. Fischer, M. Tröls, and A. Egyed, « An empirical study on the impact of inconsistency feedback during model and code co-changing. », *The Journal of Object Technology*, vol. 18, *2*, pp. 10–1, 2019.

[6] V. C. Pham, A. Radermacher, S. Gerard, and S. Li, « Bidirectional mapping between architecture model and code for synchronization », *in 2017 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2017, pp. 239–242.

[7] R. Jongeling, J. Fredriksson, F. Ciccozzi, A. Cicchetti, and J. Carlson, « Towards consistency checking between a system model and its implementation », *in International Conference on Systems Modelling and Management*, Springer, 2020, pp. 30–39.

[8] R. Jongeling, J. Fredriksson, F. Ciccozzi, J. Carlson, and A. Cicchetti, « Structural consistency between a system model and its implementation: a design science study in industry », *in The European Conference on Modelling Foundations and Applications (ECMFA)*, 2022.

[9] M. Zaheri, M. Famelis, and E. Syriani, « Towards checking consistency-breaking updates between models and generated artifacts », *in 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, IEEE, 2021, pp. 400–409.

[10] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, « Maintaining invariant traceability through bidirectional transformations », *in 2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 540–550.

[11] D. E. Khelladi, B. Combemale, M. Acher, O. Barais, and J.-M. Jézéquel, « Co-evolving code with evolving metamodels », *in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1496–1508, ISBN: 9781450371216. DOI: 10.1145/3377811.3380324. [Online]. Available: https://doi.org/10.1145/3377811.3380324.

[12] X. Ge and E. Murphy-Hill, « Manual refactoring changes with automated refactoring validation », *in Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 1095–1105, ISBN: 9781450327565. DOI: 10.1145/2568225.2568280. [Online]. Available: https://doi.org/10.1145/2568225.2568280.

[13] M. Gligoric, L. Eloussi, and D. Marinov, « Ekstazi: lightweight test selection », *in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 713–716. DOI: 10.1109/ICSE.2015.230.

[14] S. Ducasse, M. Lanza, and S. Tichelaar, « Moose: an extensible language-independent environment for reengineering object-oriented systems », *in Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, vol. 4, 2000.

[15] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le, « Chatgpt for vulnerability detection, classification, and repair: how far are we? », *arXiv preprint arXiv:2310.09810*, 2023.

[16] M. M. A. Kabir, S. A. Hassan, X. Wang, Y. Wang, H. Yu, and N. Meng, « An empirical study of chatgpt-3.5 on question answering and code maintenance », *arXiv preprint arXiv:2310.02104*, 2023.

[17] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, « Multilingual code co-evolution using large language models », *arXiv preprint arXiv:2307.14991*, 2023.

[18] K. Chen, Y. Yang, B. Chen, J. A. H. López, G. Mussbacher, and D. Varró, « Automated domain modeling with large language models: a comparative study », *in 2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2023, pp. 162–172.

[19] J. Cámara, J. Troya, L. Burgueño, and A. Vallecillo, « On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml », *Software and Systems Modeling*, pp. 1–13, 2023.

[20] M. B. Chaaben, L. Burgueño, and H. Sahraoui, « Towards using few-shot prompt learning for automating model completion », *in IEEE/ACM 45th Int. Conf. on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, IEEE, 2023, pp. 7–12.

[21] B. Selic, « The pragmatics of model-driven development », *IEEE Software*, vol. 20, *5*, pp. 19–25, 2003. DOI: 10.1109/MS.2003.1231146.

[22] J. Bézivin, « On the unification power of models », *Softw. Syst. Model.*, vol. 4, *2*, pp. 171–188, May 2005, ISSN: 1619-1366. DOI: 10.1007/s10270-005-0079-0. [Online]. Available: https://doi.org/10.1007/s10270-005-0079-0.

[23] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani, « Mde adoption : challenges and success criteria », vol. 5421, 2009, pp. 54–59, ISBN: 9783642016479. DOI: 10.1007/978-3-642-01648-6_6.

[24] P. Mohagheghi and V. Dehlen, « Where is the proof?-a review of experiences from applying mde in industry », *in Model Driven Architecture–Foundations and Applications*, Springer, 2008, pp. 432–443.

[25] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, « Modeling languages in industry 4.0: an extended systematic mapping study », *Software and Systems Modeling*, vol. 19, pp. 67–94, 2020.

[26] J. Hutchinson, M. Rouncefield, and J. Whittle, « Model-driven engineering practices in industry », *in Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 633–642, ISBN: 9781450304450. DOI: 10.1145/1985793.1985882. [Online]. Available: https://doi.org/10.1145/1985793.1985882.

[27] T. Stahl, M. Völter, and K. Czarnecki, *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006.

[28] E. Seidewitz, « What models mean », *IEEE software*, vol. 20, *5*, p. 26, 2003.

[29] M. Volter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[30] D. Gašević, N. Kaviani, and M. Hatala, « On metamodeling in megamodels », *in Model Driven Engineering Languages and Systems: 10th International Conference, MoDELS 2007, Nashville, USA, September 30-October 5, 2007. Proceedings 10*, Springer, 2007, pp. 91–105.

[31] J. de Lara and E. Guerra, « Domain-specific textual meta-modelling languages for model driven engineering », *in Modelling Foundations and Applications: 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings 8*, Springer, 2012, pp. 259–274.

[32] B. Lientz and E. Swanson, *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980, ISBN: 9780201042054. [Online]. Available: `https://books.google.fr/books?id=8a6gAAAAMAAJ`.

[33] E. B. Swanson, « The dimensions of maintenance », *Proceedings - International Conference on Software Engineering*, pp. 492–497, 1976, ISSN: 02705257.

[34] K. H. Bennett and V. T. Rajlich, « Software maintenance and evolution: a roadmap », *in Proceedings of the Conference on the Future of Software Engineering*, ACM, 2000, pp. 73–87.

[35] N. F. Schneidewind, « The state of software maintenance », *IEEE Transactions on Software Engineering*, vol. 13, *3*, p. 303, 1987.

[36] T. Mens and T. Tourwé, « A survey of software refactoring », *Software Engineering, IEEE Transactions on*, vol. 30, *2*, pp. 126–139, 2004.

[37] W. Kessentini, H. Sahraoui, and M. Wimmer, « Automated metamodel/model co-evolution using a multi-objective optimization approach », *in 12th ECMFA*, 2016.

[38] A. M. Eilertsen and A. H. Bagge, « Exploring api/client co-evolution », *in 2018 IEEE/ACM 2nd International Workshop on API Usage and Evolution (WAPI)*, 2018, pp. 10–13.

[39] M. Herrmannsdoerfer, S. Benz, and E. Juergens, « Cope-automating coupled evolution of metamodels and models. », *in ECOOP*, Springer, vol. 9, 2009, pp. 52–76.

[40] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, « Attention is all you need », *in Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010, ISBN: 9781510860964.

[41] C. Zhou, Q. Li, C. Li, *et al.*, « A comprehensive survey on pretrained foundation models: A history from BERT to chatgpt », *CoRR*, vol. abs/2302.09419, 2023. DOI: 10.48550/ARXIV.2302.09419. arXiv: 2302.09419. [Online]. Available: https://doi.org/10.48550/arXiv.2302.09419.

[42] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: 1810.04805 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1810.04805.

[43] M. Lewis, Y. Liu, N. Goyal, *et al.*, *Bart: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension*, 2019. arXiv: 1910.13461 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1910.13461.

[44] T. Wu, S. He, J. Liu, *et al.*, « A brief overview of chatgpt: the history, status quo and potential future development », *IEEE/CAA Journal of Automatica Sinica*, vol. 10, *5*, pp. 1122–1136, 2023. DOI: 10.1109/JAS.2023.123618.

[45] D. Yokoyama, K. Nishiura, and A. Monden, « Identifying security bugs in issue reports: comparison of bert, n-gram idf and chatgpt », *in 2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA)*, 2024, pp. 328–333. DOI: 10.1109/SERA61261.2024.10685583.

[46] S. Frieder, L. Pinchetti, A. Chevalier, *et al.*, « Mathematical capabilities of chatgpt », *in Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23, New Orleans, LA, USA: Curran Associates Inc., 2024.

[47] T. B. Brown, B. Mann, N. Ryder, *et al.*, « Language models are few-shot learners », *in Proceedings of the 34th International Conference on Neural Information Process-*

ing Systems, ser. NIPS '20, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.

[48] A. Chowdhery, S. Narang, J. Devlin, *et al.*, « Palm: scaling language modeling with pathways », *J. Mach. Learn. Res.*, vol. 24, *1*, Mar. 2024, ISSN: 1532-4435.

[49] J. Hoffmann, S. Borgeaud, A. Mensch, *et al.*, « Training compute-optimal large language models », *in Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22, New Orleans, LA, USA: Curran Associates Inc., 2024, ISBN: 9781713871088.

[50] T. B. Brown, B. Mann, N. Ryder, *et al.*, *Language models are few-shot learners*, 2020. arXiv: 2005.14165 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2005.14165.

[51] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, « Pre-train, prompt, and predict: a systematic survey of prompting methods in natural language processing », *ACM Comput. Surv.*, vol. 55, *9*, Jan. 2023, ISSN: 0360-0300. DOI: 10.1145/3560815. [Online]. Available: https://doi.org/10.1145/3560815.

[52] J. Wei, X. Wang, D. Schuurmans, *et al.*, « Chain-of-thought prompting elicits reasoning in large language models », *in Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22, New Orleans, LA, USA: Curran Associates Inc., 2024, ISBN: 9781713871088.

[53] T. Mens, *Introduction and roadmap: History and challenges of software evolution*. Springer, 2008.

[54] R. Hebig, D. E. Khelladi, and R. Bendraou, « Approaches to co-evolution of metamodels and models: a survey », *IEEE Transactions on Software Engineering*, vol. 43, *5*, pp. 396–414, 2016.

[55] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, « An extensive catalog of operators for the coupled evolution of metamodels and models », *in Software Language Engineering*, Malloy, Staab, and Brand, Eds., Springer, Jan. 2011, pp. 163–182, ISBN: 978-3-642-19439-9, 978-3-642-19440-5.

[56] B. Gruschko, D. Kolovos, and R. Paige, « Towards synchronizing models with evolving metamodels », *in Proceedings of the International Workshop on Model-Driven Software Evolution*, Amsterdam, Netherlands, 2007, p. 3.

[57] D. Dig and R. Johnson, « How do apis evolve? a story of refactoring », *Journal of software maintenance and evolution: Research and Practice*, vol. 18, *2*, pp. 83–107, 2006.

[58] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, « An extensive catalog of operators for the coupled evolution of metamodels and models », *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6563 LNCS, pp. 163–182, 2011, ISSN: 03029743. DOI: 10.1007/978-3-642-19440-5_10.

[59] S. D. Vermolen, G. Wachsmuth, and E. Visser, « Reconstructing complex metamodel evolution », *in Software Language Engineering*, Springer, 2012, pp. 201–221.

[60] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais, « Detecting complex changes during metamodel evolution », *in CAISE*, Springer, 2015, pp. 263–278.

[61] S. Alter, « Work system theory: A bridge between business and IT views of systems », *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9097, pp. 520–521, 2015, ISSN: 16113349. DOI: 10.1007/978-3-319-19069-3.

[62] J. R. Williams, R. F. Paige, and F. A. Polack, « Searching for model migration strategies », *in Proceedings of the 6th International Workshop on Models and Evolution*, ACM, 2012, pp. 39–44.

[63] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, « Managing dependent changes in coupled evolution », *in Theory and Practice of Model Transformations*, Springer, 2009, pp. 35–51.

[64] P. Langer, M. Wimmer, P. Brosch, *et al.*, « A posteriori operation detection in evolving software models », *Journal of Systems and Software*, vol. 86, *2*, pp. 551–566, 2013.

[65] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M. P. Gervais, « Detecting complex changes and refactorings during (Meta)model evolution », *Information Systems*, vol. 62, pp. 220–241, 2016, ISSN: 03064379. DOI: 10.1016/j.is.2016.05.002. [Online]. Available: http://dx.doi.org/10.1016/j.is.2016.05.002.

[66] L. Bettini, D. Di Ruscio, L. Iovino, and A. Pierantonio, « An executable metamodel refactoring catalog », *Software and Systems Modeling*, vol. 21, *5*, pp. 1689–1709, 2022.

[67] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, « Constraint-driven modeling through transformation », *Software & Systems Modeling*, vol. 14, *2*, pp. 573–596, 2015.

[68] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais, « Detecting complex changes and refactorings during (meta) model evolution », *Information Systems*, 2016.

[69] D. Di Ruscio, L. Iovino, and A. Pierantonio, « What is needed for managing co-evolution in mde? », *in Proceedings of the 2nd International Workshop on Model Comparison in Practice*, ser. IWMCP '11, Zurich, Switzerland: Association for Computing Machinery, 2011, pp. 30–38, ISBN: 9781450306683. DOI: 10.1145/2000410.2000416. [Online]. Available: https://doi.org/10.1145/2000410.2000416.

[70] T. Levendovszky, D. Balasubramanian, A. Narayanan, F. Shi, C. van Buskirk, and G. Karsai, « A semi-formal description of migrating domain-specific models with evolving domains », *Software & Systems Modeling*, vol. 13, *2*, pp. 807–823, 2014.

[71] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin, « Managing model adaptation by precise detection of metamodel changes », *in Model Driven Architecture-Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings 5*, Springer, 2009, pp. 34–49.

[72] J. Garcia, O. Diaz, and M. Azanza, « Model transformation co-evolution: a semi-automatic approach », *in International conference on software language engineering*, Springer, 2012, pp. 144–163.

[73] Z. Xing and E. Stroulia, « Refactoring detection based on umldiff change-facts queries », *in Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, IEEE, 2006, pp. 263–274.

[74] I. H. Moghadam and M. O. Cinneide, « Automated refactoring using design differencing », *in Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, IEEE, 2012, pp. 43–52.

[75] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. Lara, « Example-driven meta-model development », *Softw. Syst. Model.*, vol. 14, *4*, pp. 1323–1347, Oct. 2015, ISSN: 1619-1366. DOI: 10.1007/s10270-013-0392-y. [Online]. Available: https://doi.org/10.1007/s10270-013-0392-y.

[76] A. Gargantini, E. Riccobene, and P. Scandurra, « A semantic framework for metamodel-based languages », *Automated Software Engg.*, vol. 16, *3–4*, pp. 415–454, Dec. 2009, ISSN: 0928-8910. DOI: 10.1007/s10515-009-0053-0. [Online]. Available: https://doi.org/10.1007/s10515-009-0053-0.

[77] R. Hebig, D. E. Khelladi, and R. Bendraou, *Approaches to co-evolution of metamodels and models: A survey*, May 2017. DOI: 10.1109/TSE.2016.2610424.

[78] S. Vermolen and E. Visser, « Heterogeneous coupled evolution of software languages », *in Model Driven Engineering Languages and Systems*, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 630–644, ISBN: 978-3-540-87875-9.

[79] J. Sprinkle and G. Karsai, « A domain-specific visual language for domain model evolution », *Journal of Visual Languages & Computing*, vol. 15, *3-4*, pp. 291–307, 2004.

[80] M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel, « On using inplace transformations for model co-evolution », *in Proc. 2nd int. workshop model transformation with atl*, vol. 711, 2010, pp. 65–78.

[81] D. Wagelaar, L. Iovino, D. Di Ruscio, and A. Pierantonio, « Translational semantics of a co-evolution specific language with the emf transformation virtual machine », *in Theory and Practice of Model Transformations*, Z. Hu and J. de Lara, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 192–207, ISBN: 978-3-642-30476-7.

[82] C. Krause, J. Dyck, and H. Giese, « Metamodel-specific coupled evolution based on dynamically typed graph transformations », *in Theory and Practice of Model Transformations*, K. Duddy and G. Kappel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 76–91, ISBN: 978-3-642-38883-5.

[83] T. Levendovszky, D. Balasubramanian, A. Narayanan, F. Shi, C. Buskirk, and G. Karsai, « A semi-formal description of migrating domain-specific models with evolving domains », *Softw. Syst. Model.*, vol. 13, *2*, pp. 807–823, May 2014, ISSN:

1619-1366. DOI: 10.1007/s10270-012-0313-5. [Online]. Available: https://doi.org/10.1007/s10270-012-0313-5.

[84] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. Polack, and S. Poulding, « Epsilon flock: a model migration language », *Softw. Syst. Model.*, vol. 13, *2*, pp. 735–755, May 2014, ISSN: 1619-1366. DOI: 10.1007/s10270-012-0296-2. [Online]. Available: https://doi.org/10.1007/s10270-012-0296-2.

[85] M. D. Del Fabro and P. Valduriez, « Semi-automatic model integration using matching transformations and weaving models », *in Proceedings of the 2007 ACM symposium on Applied computing*, 2007, pp. 963–970.

[86] G. de Geest, S. Vermolen, A. van Deursen, and E. Visser, « Generating version convertors for domain-specific languages », *in 2008 15th Working Conference on Reverse Engineering*, IEEE, 2008, pp. 197–201.

[87] B. Meyers, M. Wimmer, A. Cicchetti, and J. Sprinkle, « A generic in-place transformation-based approach to structured model co-evolution », *Electronic Communications of the EASST*, vol. 42, 2011.

[88] F. Anguel, A. Amirat, and N. Bounour, « Using weaving models in metamodel and model co-evolution approach », *in 2014 6th International Conference on Computer Science and Information Technology (CSIT)*, IEEE, 2014, pp. 142–147.

[89] J. Hößler, M. Soden, H. Eichler, *et al.*, « Coevolution of models, metamodels and transformations », *Models and Human Reasoning*, pp. 129–154, 2005.

[90] H. Florez, M. Sánchez, J. Villalobos, and G. Vega, « Coevolution assistance for enterprise architecture models », *in Proceedings of the 6th International Workshop on Models and Evolution*, 2012, pp. 27–32.

[91] H. A. F. Fernandez *et al.*, « Adapting models in metamodels composition processes », *Revista Vinculos*, vol. 10, *1*, pp. 96–108, 2013.

[92] G. Wachsmuth, « Metamodel adaptation and model co-adaptation », *in ECOOP*, Springer, vol. 7, 2007, pp. 600–624.

[93] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, « Managing dependent changes in coupled evolution », *in International Conference on Theory and Practice of Model Transformations*, Springer, 2009, pp. 35–51.

[94] M. Van Den Brand, Z. Protić, and T. Verhoeff, « A generic solution for syntax-driven model co-evolution », *in Objects, Models, Components, Patterns: 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings 49*, Springer, 2011, pp. 36–51.

[95] S. Becker, B. Gruschko, T. Goldschmidt, and H. Koziolek, « A process model and classification scheme for semi-automatic meta-model evolution », *in 1st Workshop MDD, SOA und IT-Management (MSI), GI, GiTO-Verlag*, 2007, pp. 35–46.

[96] M. Herrmannsdoerfer, « Operation-based versioning of metamodels with cope », *in Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, IEEE, 2009, pp. 49–54.

[97] H. Wittern, « Determining the necessity of human intervention when migrating models of an evolved dsl », *in 2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops*, IEEE, 2013, pp. 209–218.

[98] F. Anguel, A. Amirat, and N. Bounour, « Towards models and metamodels co-evolution approach », *in 2013 11th International Symposium on Programming and Systems (ISPS)*, IEEE, 2013, pp. 163–167.

[99] A. Demuth, M. Riedl-Ehrenleitner, R. E. Lopez-Herrejon, and A. Egyed, « Co-evolution of metamodels and models through consistent change propagation », *Journal of Systems and Software*, vol. 111, pp. 281–297, 2016.

[100] P. Gómez, M. E. Sánchez, H. Florez, and J. Villalobos, « An approach to the co-creation of models and metamodels in enterprise architecture projects. », *J. Object Technol.*, vol. 13, *3*, pp. 2–1, 2014.

[101] J. Schönböck, A. Kusel, J. Etzlstorfer, *et al.*, « Care: a constraint-based approach for re-establishing conformance-relationships », *in Proceedings of the Tenth Asia-Pacific Conference on Conceptual Modelling-Volume 154*, 2014, pp. 19–28.

[102] W. Kessentini, H. Sahraoui, and M. Wimmer, « Automated metamodel/model co-evolution using a multi-objective optimization approach », *in Modelling Foundations and Applications: 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings 12*, Springer, 2016, pp. 138–155.

[103] W. Kessentini, H. Sahraoui, and M. Wimmer, « Automated metamodel/model co-evolution: a search-based approach », *Information and Software Technology*, vol. 106, pp. 49–67, 2019.

[104] W. Kessentini and V. Alizadeh, « Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search », *in Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 68–78.

[105] D. Di Ruscio, L. Iovino, and A. Pierantonio, « Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems », *in Proceedings of the 6th International Conference on Graph Transformations*, ser. ICGT'12, Bremen, Germany: Springer-Verlag, 2012, pp. 20–37, ISBN: 9783642336539. DOI: 10.1007/978-3-642-33654-6_2. [Online]. Available: https://doi.org/10.1007/978-3-642-33654-6_2.

[106] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, « Automating co-evolution in model-driven engineering », *in 2008 12th International IEEE enterprise distributed object computing conference*, IEEE, 2008, pp. 222–231.

[107] K. Garcés, J. M. Vara, F. Jouault, and E. Marcos, « Adapting transformations to metamodel changes via external transformation composition », *Software & Systems Modeling*, vol. 13, *2*, pp. 789–806, 2014.

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

[109] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, « Supporting the co-evolution of metamodels and constraints through incremental constraint management », *in Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*, Berlin, Heidelberg: Springer-Verlag, 2013, pp. 287–303, ISBN: 9783642415326. DOI: 10.1007/978-3-642-41533-3_18. [Online]. Available: https://doi.org/10.1007/978-3-642-41533-3_18.

[110] S. Marković and T. Baar, « Refactoring ocl annotated uml class diagrams », *Software & Systems Modeling*, vol. 7, pp. 25–47, 2008.

[111] K. Hassam, S. Sadou, V. L. Gloahec, and R. Fleurquin, « Assistance system for ocl constraints adaptation during metamodel evolution », *in Software Maintenance and Reengineering (CSMR), 15th European Conference on*, IEEE, 2011, pp. 151–160.

[112] A. Kusel, J. Etzlstorfer, E. Kapsammer, *et al.*, « A systematic taxonomy of metamodel evolution impacts on ocl expressions. », *in ME@ MoDELS*, 2014, pp. 2–11.

[113] J. Cabot and J. Conesa, « Automatic integrity constraint evolution due to model subtract operations », *in Conceptual Modeling for Advanced Application Domains*, Springer, 2004, pp. 350–362.

[114] D. E. Khelladi, R. Bendraou, R. Hebig, and M.-P. Gervais, « A semi-automatic maintenance and co-evolution of ocl constraints with (meta) model evolution », *Journal of Systems and Software*, vol. 134, pp. 242–260, 2017.

[115] E. Batot, W. Kessentini, H. Sahraoui, and M. Famelis, « Heuristic-based recommendation for metamodel — ocl coevolution », *in 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2017, pp. 210–220. DOI: 10.1109/MODELS.2017.25.

[116] E. Cherfa, S. Mesli-Kesraoui, C. Tibermacine, S. Sadou, and R. Fleurquin, « Identifying metamodel inaccurate structures during metamodel/constraint co-evolution », *in 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2021, pp. 24–34.

[117] E. Guerra, J. Lara, M. Wimmer, *et al.*, « Automated verification of model transformations based on visual contracts », *Automated Software Engg.*, vol. 20, *1*, pp. 5–46, Mar. 2013, ISSN: 0928-8910. DOI: 10.1007/s10515-012-0102-y. [Online]. Available: https://doi.org/10.1007/s10515-012-0102-y.

[118] E. Batot, W. Kessentini, H. Sahraoui, and M. Famelis, « Heuristic-based recommendation for metamodel—ocl coevolution », *in 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2017, pp. 210–220.

[119] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, W. Schwinger, and J. Schonbock, « Consistent co-evolution of models and transformations », *in ACM/IEEE 18th MODELS*, 2015, pp. 116–125.

[120] D. E. Khelladi, R. Kretschmer, and A. Egyed, « Change propagation-based and composition-based co-evolution of transformations with evolving metamodels », *in Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ACM, 2018, pp. 404–414.

[121] D. Di Ruscio, L. Iovino, and A. Pierantonio, « What is needed for managing co-evolution in mde? », *in Proceedings of the 2nd International Workshop on Model Comparison in Practice*, ACM, 2011, pp. 30–38.

[122] D. Mendez, A. Etien, A. Muller, and R. Casallas, « Towards transformation migration after metamodel evolution », *ME Wokshop@MODELS*, 2010.

[123] W. Kessentini, H. Sahraoui, and M. Wimmer, « Automated co-evolution of metamodels and transformation rules: a search-based approach », *in International Symposium on Search Based Software Engineering*, Springer, 2018, pp. 229–245.

[124] J. García, O. Diaz, and M. Azanza, « Model transformation co-evolution: a semi-automatic approach », *in Software Language Engineering*, K. Czarnecki and G. Hedin, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 144–163, ISBN: 978-3-642-36089-3.

[125] A. Kusel, J. Etzlstorfer, E. Kapsammer, *et al.*, « Systematic co-evolution of ocl expressions », *in 11th APCCM 2015*, vol. 27, 2015, p. 30.

[126] A. Correa and C. Werner, « Refactoring object constraint language specifications », *Software & Systems Modeling*, vol. 6, *2*, pp. 113–138, 2007.

[127] J. Henkel and A. Diwan, « Catchup! capturing and replaying refactorings to support api evolution », *in Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, IEEE, 2005, pp. 274–283.

[128] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, « A graph-based approach to api usage adaptation », *ACM Sigplan Notices*, vol. 45, *10*, pp. 302–321, 2010.

[129] B. Dagenais and M. P. Robillard, « Recommending adaptive changes for framework evolution », *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, *4*, p. 19, 2011.

[130] B. Dagenais and M. P. Robillard, « Semdiff: analysis and recommendation support for api evolution », *in 2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 599–602. DOI: 10.1109/ICSE.2009.5070565.

[131] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, « A graph-based approach to api usage adaptation », *SIGPLAN Not.*, vol. 45, *10*, pp. 302–321, Oct. 2010, ISSN: 0362-1340. DOI: 10.1145/1932682.1869486. [Online]. Available: https://doi.org/10.1145/1932682.1869486.

[132] J. Andersen and J. L. Lawall, « Generic patch inference », *Automated software engineering*, vol. 17, *2*, pp. 119–148, 2010.

[133] S. Gerasimou, M. Kechagia, D. Kolovos, R. Paige, and G. Gousios, « On software modernisation due to library obsolescence », *in Proceedings of the 2nd International Workshop on API Usage and Evolution*, ser. WAPI '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 6–9, ISBN: 9781450357548. DOI: 10.1145/3194793.3194798. [Online]. Available: https://doi.org/10.1145/3194793.3194798.

[134] O. Zaitsev, S. Ducasse, N. Anquetil, and A. Thiefaine, « How libraries evolve: a survey of two industrial companies and an open-source community », *in 2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, 2022, pp. 309–317. DOI: 10.1109/APSEC57359.2022.00043.

[135] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, « An empirical study on the impact of refactoring activities on evolving client-used apis », *Inf. Softw. Technol.*, vol. 93, *C*, pp. 186–199, Jan. 2018, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2017.09.007. [Online]. Available: https://doi.org/10.1016/j.infsof.2017.09.007.

[136] R. Robbes, M. Lungu, and D. Röthlisberger, « How do developers react to api deprecation? the case of a smalltalk ecosystem », *in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, Cary, North Carolina: Association for Computing Machinery, 2012, ISBN: 9781450316149. DOI: 10.1145/2393596.2393662. [Online]. Available: https://doi.org/10.1145/2393596.2393662.

[137] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. Tulio Valente, « How do developers react to api evolution? the pharo ecosystem case », *in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 251–260. DOI: 10.1109/ICSM.2015.7332471.

[138] A. A. Sawant, R. Robbes, and A. Bacchelli, « On the reaction to deprecation of 25,357 clients of 4+1 popular java apis », *in 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 400–410. DOI: 10.1109/ICSME.2016.64.

[139] L. Xavier, A. Brito, A. Hora, and M. T. Valente, « Historical and impact analysis of api breaking changes: a large-scale study », *in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 138–147. DOI: 10.1109/SANER.2017.7884616.

[140] A. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse, « How do developers react to api evolution? a large-scale empirical study », *Software Quality Journal*, vol. 26, *1*, pp. 161–191, Mar. 2018, ISSN: 0963-9314. DOI: 10.1007/s11219-016-9344-4. [Online]. Available: https://doi.org/10.1007/s11219-016-9344-4.

[141] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, « Do developers update their library dependencies? », *Empirical Softw. Engg.*, vol. 23, *1*, pp. 384–417, Feb. 2018, ISSN: 1382-3256. DOI: 10.1007/s10664-017-9521-5. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5.

[142] K. Jezek, J. Dietrich, and P. Brada, « How java apis break - an empirical study », *Inf. Softw. Technol.*, vol. 65, *C*, pp. 129–146, Sep. 2015, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.02.014. [Online]. Available: https://doi.org/10.1016/j.infsof.2015.02.014.

[143] O. Zaytsev, « Data Mining-based tools to support library update », Theses, Université de Lille, Oct. 2022. [Online]. Available: https://theses.hal.science/tel-03998632.

[144] S. Mostafa, R. Rodriguez, and X. Wang, « Experience paper: a study on behavioral backward incompatibilities of java software libraries », *in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, Santa Barbara, CA, USA: Association for Computing Machinery, 2017, pp. 215–225, ISBN: 9781450350761. DOI: 10.1145/3092703.3092721. [Online]. Available: https://doi.org/10.1145/3092703.3092721.

[145] N. Meng, M. Kim, and K. S. McKinley, « Lase: locating and applying systematic edits by learning from examples », *in 2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 502–511. DOI: 10.1109/ICSE.2013.6606596.

[146] M. Fazzini, Q. Xin, and A. Orso, « Apimigrator: an api-usage migration tool for android apps », *in Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '20, Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 77–80, ISBN: 9781450379595. DOI: 10.1145/3387905.3388608. [Online]. Available: https://doi.org/10.1145/3387905.3388608.

[147] M. Lamothe, W. Shang, and T.-H. P. Chen, « A3: assisting android api migrations using code examples », *IEEE Transactions on Software Engineering*, vol. 48, *2*, pp. 417–431, 2022. DOI: 10.1109/TSE.2020.2988396.

[148] S. Xu, Z. Dong, and N. Meng, « Meditor: inference and application of api migration edits », *in 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 335–346. DOI: 10.1109/ICPC.2019.00052.

[149] H. Zhong and N. Meng, « Compiler-directed migrating api callsite of client code », *in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, <conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>: Association for Computing Machinery, 2024, ISBN: 9798400702174. DOI: 10.1145/3597503.3639084. [Online]. Available: https://doi.org/10.1145/3597503.3639084.

[150] J. Di Rocco, P. T. Nguyen, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, « Deepmig: a transformer-based approach to support coupled library and code migrations », *Information and Software Technology*, vol. 177, p. 107 588, 2025, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2024.107588. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924001939.

[151] C. L. Goues, M. Pradel, and A. Roychoudhury, « Automated program repair », *Communications of the ACM*, vol. 62, *12*, pp. 56–65, 2019.

[152] K. Liu, S. Wang, A. Koyuncu, *et al.*, « On the efficiency of test suite based program repair: a systematic assessment of 16 automated repair systems for java programs », *in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.

[153] M. Monperrus, « Automatic software repair: a bibliography », *ACM Computing Surveys (CSUR)*, vol. 51, *1*, p. 17, 2018.

[154] L. Gazzola, D. Micucci, and L. Mariani, « Automatic software repair: a survey », *IEEE Transactions on Software Engineering*, vol. 45, *1*, pp. 34–67, 2017.

[155] N. Meng, M. Kim, and K. S. McKinley, « Lase: locating and applying systematic edits by learning from examples », *in Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 502–511, ISBN: 9781467330763.

[156] C. S. Xia, Y. Wei, and L. Zhang, « Automated program repair in the era of large pre-trained language models », *in Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 1482–1494, ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00129. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129.

[157] H. Ruan, H. Nguyen, R. Shariffdeen, Y. Noller, and A. Roychoudhury, « Evolutionary testing for program repair », *in 2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 105–116. DOI: 10.1109/ICST60714.2024.00058. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST60714.2024.00058.

[158] G. Fraser and A. Arcuri, « Evosuite: automatic test suite generation for object-oriented software », *in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[159] C. S. Xia and L. Zhang, « Automated program repair via conversation: fixing 162 out of 337 bugs for 0.42*EachusingChatGPT* », *in Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024, Vienna, Austria: Association for Computing Machinery, 2024, pp. 819–831, ISBN: 9798400706127. DOI: 10.1145/3650212.3680323. [Online]. Available: https://doi.org/10.1145/3650212.3680323.

[160] L. Chen, Y. Pei, M. Pan, T. Zhang, Q. Wang, and C. A. Furia, « Program repair with repeated learning », *IEEE Transactions on Software Engineering*, vol. 49, *2*, pp. 831–848, 2023. DOI: 10.1109/TSE.2022.3164662.

[161] I. Ivanov, J. Bézivin, and M. Aksit, « Technological spaces: an initial appraisal », *in 4th International Symposium on Distributed Objects and Applications, DOA 2002*, 2002.

[162] J.-M. Favre, « Meta-model and model co-evolution within the 3d software space », *ELISA*, vol. 3, pp. 98–109, 2003.

[163] R. Lammel and V. Zaytsev, « Recovering grammar relationships for the java language specification », *in 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 178–186. DOI: 10.1109/SCAM.2009.29.

[164] R. Lammel and W. Lohmann, *Format evolution*. Citeseer, 2001.

[165] B. Meyer, « Schema evolution: concepts, terminology, and solutions », *Computer*, vol. 29, *10*, pp. 119–121, 1996.

[166] G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis, and G. Antoniou, « Ontology change: classification and survey », *The Knowledge Engineering Review*, vol. 23, *2*, pp. 117–152, 2008.

[167] M. Schuts, M. Alonso, and J. Hooman, « Industrial experiences with the evolution of a dsl », *in Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, 2021, pp. 21–30.

[168] J.-M. Favre, « Languages evolve too! changing the software time scale », *in Eighth International Workshop on Principles of Software Evolution IWPSE*, IEEE, 2005, pp. 33–44.

[169] M. Herrmannsdörfer and G. Wachsmuth, « Coupled evolution of software metamodels and models », *in Evolving Software Systems*, Springer, 2013, pp. 33–63.

[170] J. Mengerink, R. R. Schiffelers, A. Serebrenik, and M. van den Brand, « Dsl/model co-evolution in industrial emf-based mdse ecosystems. », *in ME@ MoDELS*, 2016, pp. 2–7.

[171] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth, « Language evolution in practice: the history of gmf », *in Software Language Engineering*, M. van den Brand, D. Gašević, and J. Gray, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 3–22, ISBN: 978-3-642-12107-4.

[172] R.-G. Urma, « Programming language evolution », University of Cambridge, Computer Laboratory, Tech. Rep., 2017.

[173] J. Dietrich, K. Jezek, and P. Brada, « What java developers know about compatibility, and why this matters », *Empirical Software Engineering*, vol. 21, pp. 1371–1396, 2016.

[174] R. S. Arnold, *Software change impact analysis*. IEEE Computer Society Press, 1996.

[175] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, « Chianti: a tool for change impact analysis of java programs », *in Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2004, pp. 432–448.

[176] B. G. Ryder and F. Tip, « Change impact analysis for object-oriented programs », *in Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 46–53.

[177] L. Ochoa, T. Degueule, and J.-R. Falleri, « Breakbot: analyzing the impact of breaking changes to assist library evolution », *in Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 26–30, ISBN: 9781450392242. DOI: 10.1145/3510455.3512783. [Online]. Available: https://doi.org/10.1145/3510455.3512783.

[178] M. Zhang, K. Ogata, and K. Futatsugi, « Formalization and verification of behavioral correctness of dynamic software updates », *Electronic Notes in Theoretical Computer Science*, vol. 294, pp. 12–23, 2013, Proceedings of the 2013 Validation Strategies for Software Evolution (VSSE) Workshop, ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2013.02.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066113000157.

[179] Y. Qi, W. Liu, W. Zhang, and D. Yang, « How to measure the performance of automated program repair », *in 2018 5th International Conference on Information Science and Control Engineering (ICISCE)*, 2018, pp. 246–250. DOI: 10.1109/ICISCE.2018.00059.

[180] K. Liu, L. Li, A. Koyuncu, *et al.*, « A critical review on the evaluation of automated program repair systems », *Journal of Systems and Software*, vol. 171, p. 110 817, 2021, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110817. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302156.

[181] W. F. Opdyke, « Refactoring object-oriented frameworks », UMI Order No. GAX93-05645, Ph.D. dissertation, USA, 1992.

[182] G. Soares, R. Gheyi, T. Massoni, M. Cornélio, and D. Cavalcanti, « Generating unit tests for checking refactoring safety », *in Brazilian Symposium on Programming Languages*, vol. 1175, 2009, pp. 159–172.

[183] M. Wahler, U. Drofenik, and W. Snipes, « Improving code maintainability: a case study on the impact of refactoring », *in 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 493–501. DOI: 10.1109/ICSME.2016.52.

[184] L. Da Silva, P. Borba, T. Maciel, *et al.*, « Detecting semantic conflicts with unit tests », *Journal of Systems and Software*, vol. 214, p. 112 070, 2024, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112070. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224001158.

[185] B. Roziere, J. M. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, « Leveraging automated unit tests for unsupervised code translation », *arXiv preprint arXiv:2110.06773*, 2021.

[186] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, « Asleep at the keyboard? assessing the security of github copilot's code contributions », *in 2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 754–768.

[187] D. Sobania, M. Briesch, and F. Rothlauf, « Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming », *in Proceedings of the genetic and evolutionary computation conference*, 2022, pp. 1019–1027.

[188] A. Ziegler, E. Kalliamvakou, X. A. Li, *et al.*, « Productivity assessment of neural code completion », *in ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.

[189] P. Vaithilingam, T. Zhang, and E. L. Glassman, « Expectation vs. experience: evaluating the usability of code generation tools powered by large language models », *in Chi conference on human factors in computing systems, extended abstracts*, 2022, pp. 1–7.

[190] N. Nguyen and S. Nadi, « An empirical evaluation of github copilot's code suggestions », *in The 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[191] J.-B. Döderlein, M. Acher, D. E. Khelladi, and B. Combemale, « Piloting copilot and codex: hot temperature, cold prompts, or black magic? », *arXiv preprint arXiv:2210.14699*, 2022.

[192] N. Nathalia, A. Paulo, and C. Donald, « Artificial intelligence vs. software engineers: an empirical study on performance and efficiency using chatgpt », *in The 33rd Annual International Conference on Computer Science and Software Engineering*, 2023, pp. 24–33.

[193] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, « Evaluating the code quality of ai-assisted code generation tools: an empirical study on github copilot, amazon codewhisperer, and chatgpt », *arXiv preprint arXiv:2304.10778*, 2023.

[194] Q. Guo, J. Cao, X. Xie, *et al.*, « Exploring the potential of chatgpt in automated code refinement: an empirical study », *arXiv preprint arXiv:2309.08221*, 2023.

[195] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, « Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design », *in Generative AI for Effective Software Development*, A. Nguyen-Duc, P. Abrahamsson, and F. Khomh, Eds. Cham: Springer Nature Switzerland, 2024, pp. 71–108, ISBN: 978-3-031-55642-5. DOI: 10.1007/978-3-031-55642-5_4. [Online]. Available: https://doi.org/10.1007/978-3-031-55642-5_4.

[196] G. Sridhara, S. Mazumdar, *et al.*, « Chatgpt: a study on its utility for ubiquitous software engineering tasks », *arXiv preprint arXiv:2305.16837*, 2023.

[197] D. Silva, N. Tsantalis, and M. T. Valente, « Why we refactor? confessions of github contributors », *in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 858–870, ISBN: 9781450342186. DOI: 10.1145/2950290.2950305. [Online]. Available: https://doi.org/10.1145/2950290.2950305.

[198] A. R. Sadik, A. Ceravola, F. Joublin, and J. Patra, « Analysis of chatgpt on source code », *arXiv preprint arXiv:2306.00597*, 2023.

[199] E. Hemberg, S. Moskal, and U.-M. O'Reilly, « Evolving code with a large language model », *Genetic Programming and Evolvable Machines*, vol. 25, *2*, p. 21, 2024.

[200] B. Zhang, P. Liang, Q. Feng, Y. Fu, and Z. Li, « Copilot-in-the-loop: fixing code smells in copilot-generated python code using copilot », *in Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2230–2234.

[201] S. Abukhalaf, M. Hamdaqa, and F. Khomh, « On codex prompt engineering for OCL generation: an empirical study », *in 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 148–157.

[202] S. Jiang, Y. Wang, and Y. Wang, « Selfevolve: a code evolution framework via large language models », *arXiv preprint arXiv:2306.02907*, 2023.

[203] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, « Aura: a hybrid approach to identify framework evolution », *in 2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, 2010, pp. 325–334. DOI: 10.1145/1806799.1806848.

[204] MDT, *Model development tools. modisco.* http://www.eclipse.org/modeling/mdt/?project=modisco, 2015.

[205] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, « Modisco: a generic and extensible framework for model driven reverse engineering », *in Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 173–174.

[206] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, « Modisco: a model driven reverse engineering framework », *Information and Software Technology*, vol. 56, *8*, pp. 1012–1032, 2014.

[207] P. Godefroid, D. Lehmann, and M. Polishchuk, « Differential regression testing for rest apis », *in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 312–323, ISBN: 9781450380089. DOI: 10.1145/3395363.3397374. [Online]. Available: https://doi.org/10.1145/3395363.3397374.

[208] D. E. Khelladi, R. Bendraou, and M.-P. Gervais, « Ad-room: a tool for automatic detection of refactorings in object-oriented models », *in ICSE Companion*, ACM, 2016, pp. 617–620.

[209] L. Iovino, A. Pierantonio, and I. Malavolta, « On the impact significance of metamodel evolution in mde. », *Journal of Object Technology*, vol. 11, *3*, pp. 3–1, 2012.

[210] D. E. Khelladi, H. H. Rodriguez, R. Kretschmer, and A. Egyed, « An exploratory experiment on metamodel-transformation co-evolution », *in Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*, IEEE, 2017, pp. 576–581.

[211] J. Garcıa, O. Diaz, and M. Azanza, « Model transformation co-evolution: a semi-automatic approach », *in Int. Conf. Software Language Engineering, SLE*, Springer, 2013, pp. 144–163.

[212] R. Hebig, D. E. Khelladi, and R. Bendraou, « Surveying the corpus of model resolution strategies for metamodel evolution », *in 2015 Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2015, pp. 135–142.

[213] R. Kretschmer, D. E. Khelladi, R. E. Lopez-Herrejon, and A. Egyed, « Consistent change propagation within models », *Software and Systems Modeling*, vol. 20, pp. 539–555, 2021.

[214] J. S. Cuadrado, E. Guerra, and J. de Lara, « Quick fixing atl transformations with speculative analysis », *Software & Systems Modeling*, pp. 1–35, 2018.

[215] D. E. Khelladi, R. Kretschmer, and A. Egyed, « Detecting and exploring side effects when repairing model inconsistencies », *in Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, 2019, pp. 113–126.

[216] M. Gruber, M. F. Roslan, O. Parry, F. Scharnböck, P. McMinn, and G. Fraser, « Do automatic test generation tools generate flaky tests? », 2023.

[217] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, « Regression tests to expose change interaction errors », *in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 334–344.

[218] MDT, *Model development tools. object constraints language (ocl).* `http://www.eclipse.org/modeling/mdt/?project=ocl`, 2015.

[219] MDT, *Model development tools. papyrus.* `http://www.eclipse.org/modeling/mdt/?project=papyrus`, 2015.

[220] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[221] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, « A snowballing literature study on test amplification », *Journal of Systems and Software*, vol. 157, p. 110 398, 2019, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.110398. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219301736.

[222] J. M. Rojas, G. Fraser, and A. Arcuri, « Seeding strategies in search-based unit test generation », *Software Testing, Verification and Reliability*, vol. 26, *5*, pp. 366–401, 2016. DOI: https://doi.org/10.1002/stvr.1601. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1601. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1601.

[223] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[224] P. McMinn, « Search-based software test data generation: a survey », *Software testing, Verification and reliability*, vol. 14, *2*, pp. 105–156, 2004.

[225] D. Beyer, « Advances in automatic software testing: test-comp 2022. », *in FASE*, 2022, pp. 321–335.

[226] M. Richters and M. Gogolla, « Validating uml models and ocl constraints », *in International Conference on the Unified Modeling Language*, Springer, 2000, pp. 265–277.

[227] H. K. Leung and L. White, « Insights into regression testing (software testing) », *in Proceedings. Conference on Software Maintenance-1989*, IEEE, 1989, pp. 60–69.

[228] S. Yoo and M. Harman, « Regression testing minimization, selection and prioritization: a survey », *Software testing, verification and reliability*, vol. 22, *2*, pp. 67–120, 2012.

[229] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, « A study of effective regression testing in practice », *in PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, IEEE, 1997, pp. 264–274.

[230] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, « Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions », *in Int. Conf. on Soft. Maintenance and Evolution, ICSME*, IEEE, 2021, pp. 206–216.

[231] A. Zaidman, B. Van Rompaey, A. Van Deursen, and S. Demeyer, « Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining », *Empirical Software Engineering*, vol. 16, pp. 325–364, 2011.

[232] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, « Fine-grained and accurate source code differencing », *in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[233] E. EMF, *Eclipse modeling framework (emf).* `https://github.com/eclipse-emf/org.eclipse.emf`, 2020.

[234] M. F. Roslan, J. M. Rojas, and P. McMinn, « An Empirical Comparison of EvoSuite and DSpot for Improving Developer-Written Test Suites with Respect to Mutation Score », *in Search-Based Software Engineering*, M. Papadakis and S. R. Vergilio, Eds., Cham: Springer International Publishing, 2022, pp. 19–34, ISBN: 978-3-031-21251-2.

[235] W. B. R. Herculano, E. L. G. Alves, and M. Mongiovi, « Generated tests in the context of maintenance tasks: a series of empirical studies », *IEEE Access*, vol. 10, pp. 121 418–121 443, 2022. DOI: `10.1109/ACCESS.2022.3222803`.

[236] OMG, *Object management group. unified modeling language (uml).* `http://www.omg.org/spec/UML/`, 2015.

[237] OMG, *Object management group. business process model and notation (bpmn).* `https://www.omg.org/spec/BPMN/2.0/About-BPMN/`, 2015.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : titre (en français).....

Mot clés : de 3 à 6 mots clefs

Résumé : Eius populus ab incunabulis primis ad usque pueritiae tempus extremum, quod annis circumcluditur fere trecentis, circummurana pertulit bella, deinde aetatem ingressus adultam post multiplices bellorum aerumnas Alpes transcendit et fretum, in iuvenem erectus et virum ex omni plaga quam orbis ambit inmensus, reportavit laureas et triumphos, iamque vergens in senium et nomine solo aliquotiens vincens ad tranquilliora vitae discessit. Hoc inmaturo interitu ipse quoque sui pertaesus excessit e vita aetatis nono anno atque vicensimo cum quadriennio imperasset. natus apud Tuscos in Massa Veternensi, patre Constantio Constantini fratre imperatoris, matreque Galla. Thalassius vero ea tempestate praefectus praetorio praesens ipse quoque adrogantis ingenii, considerans incitationem eius ad multorum augeri discrimina, non maturitate vel consiliis mitigabat, ut aliquotiens celsae potestates iras principum molliverunt, sed adversando iurgandoque cum parum congrueret, eum ad rabiem potius evibrabat, Augustum actus eius exaggerando creberrime docens, idque, incertum qua mente, ne lateret adfectans. quibus mox Caesar acrius efferatus, velut contumaciae quoddam vexillum altius erigens, sine respectu salutis alienae vel suae ad vertenda opposita instar rapidi fluminis irrevocabili impetu ferebatur. Hae duae provinciae bello quondam piratico catervis mixtae praedonum.

Title: Automatic metamodel and code co-evolution

Keywords: code, evolution, tests, LLM

Abstract: Software systems are becoming increasingly complex, leading to high maintenance costs that often exceed initial development expenses. Model-Driven Engineering (MDE) has emerged as a key approach to streamline development and improve productivity. It relies on the use of metamodels to generate various artifacts, including code, which developers later enhance with additional code to build the necessary language tooling, e.g.,editor, checker, compiler, data access layers, etc. Frameworks like Eclipse Modeling Framework (EMF) illustrate this workflow, generating Java APIs that are further extended for validation, debugging, and simulation. One of the major challenges in MDE is the metamodel evolution and its impact on the related artifacts. In this thesis, we focus on the code artifact and its co-evolution with evolving metamodel. Moreover, we aim to check the behavioral correctness of the metamodel and code co-evolution. Finally, with the emerbence of LLMs, we explore their usefulness for the metamodel and code co-evolution problem. This thesis addresses these challenges by: 1) proposing a new fully automated co-evolution approach of metamodels and code. this approach is based on the pattern matching of compiling errors to select suitable resolutions, then 2) proposing an automatic approach to check the behavioral correctness of the code co-evolution between different releases of a

language when its metamodel evolves. This approach leverages the test suites of the original and evolved versions of code. 3) the last contribution is about exploring LLMs' ability in proposing correct co-evolutions of the code when the metamodel evolves. This approach is based on prompt engineering, where we design and generate natural language prompts to best co-evolve the impacted code due to the metamodel evolution.

The three contributions were evaluated on EMF projects from OCL, Modisco, and papyrus. The evaluation shows that our automatic code co-evolution approach resolves 100% of errors, with 82% precision and 81% recall, significantly reducing manual effort. Furthermore, our second contribution for behavioral correctness checking can successfully trace the impacted tests due to metamodel changes representing 5% of the tests suit, thus isolating the relevant tests. Then, after the execution of the traced tests, the fluctuation in the number of passing, failing, or erroneous tests indicates whether the code co-evolution is correct or not.Using this approach allowed us to gain 88% in the number of tests and 84% in the execution time. Lastly, when evaluating the capacity of Chatgpt in code co-evolution, we variated the value of hyperparameter of temperature, and the structure of given prompts. We found that lower temperatures give better results with 88.7% of correctness rate. Regarding the structure of the prompt, including the abstraction gap information of the error and asking for alternative answers improves the correctness of the proposed co-evolutions.