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

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *« voir README et le site de votre école doctorale »*

Par
« Prénom NOM »

« Titre de la thèse »

« Sous-titre de la thèse »

Thèse présentée et soutenue à « Lieu », le « date »
Unité de recherche : « voir README et le site de de votre école doctorale »
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

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, [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. 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 Infinitest, EKSTAZI and Moose 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. For example Model generation and code generation. Since their appearance, 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 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 correctness using unit tests running before and after automatic code

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

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

➔ Second, we propose an approach that assist 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 allow 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 prototype implementation of an Eclipse plugin is available online (link). 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.

➔ Third, we investigate the ability of LLMs in giving correct co-evolutions in the context of metamodel and code co-evolution. The prototype of implementation of an Eclipse plugin is available online (link). 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.

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. I finish this chapter with a presentation of the evolution and co-evolution concept in the context of model driven engineering.

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. MDE includes various Model-Driven approaches to software development, including Model-Driven Architecture, Domain-Specific Modeling and Model-Integrated Computing [22]. The first appearance of MDE-like approaches started in the 80's [23]. Till today, MDE is still adopted and a lot of work is being done in academia and industry [8], [24]–[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 [26].

The metamodel represents the main artifact in MDE. 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, as well as constraints and modeling rules, but not the concrete syntax of the language.

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 [28] gives 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) 2.1.

Metamodeling must gather the whole knowledge that is required to define, precise, and deal with MDE challenges in its different tasks [26], related to other artifacts shown in Figure 2.2:

— The construction of metamodel describes the abstract syntax of target (software languages, 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 two metamodels.

— Code generation: the generation templates refer to the metamodel of the "system".

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

In the context of Domain-Specific Languages, DSMLs can be tailored via metamodeling to precisely match the domain's semantics and syntax. The concrete syntax that is, the concrete form of the textual or graphical constructs with which the modeling is done must represent the metamodel in an unambiguous way. Having graphic elements that linked directly to a familiar domain makes it easier to learn and allows domain experts to contribute, such as system engineers and experienced software architects, ensure that software systems meet user needs [29]. The metamodel is the basis for the automated, tool-supported processing of Metamodeling models. On the other hand, a suitable concrete syntax is the interface to the modeler and its quality decides what degree of readability the models have [27].

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 (ref).

In MDE, there are language workbenches that are used for language creation, such as Xtext, MetaEdit+ [26].

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 its specific tools

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

and represent an important topic of research in MDE.

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. , its impact on the development lifecycle.

2.3.1 Code Generation

One of the most important activities in MDE is the code generation activity 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/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.

Develop this paragraph?

2.3.2 Evolution in the MDE context

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. 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.

It is unavoidable to change, whether to answer to requirement modifications and/or technological progress. 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 that could be applied on a software.

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 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] .

Coupled evolution [39]

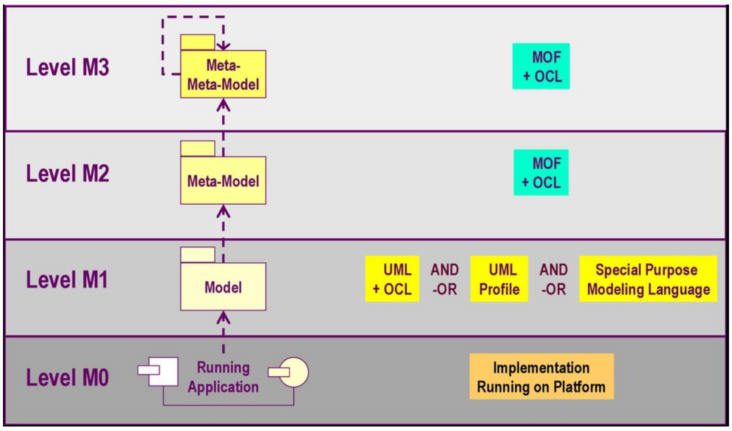

Figure 2.1 – MDA's Modeling Level Hierarchy

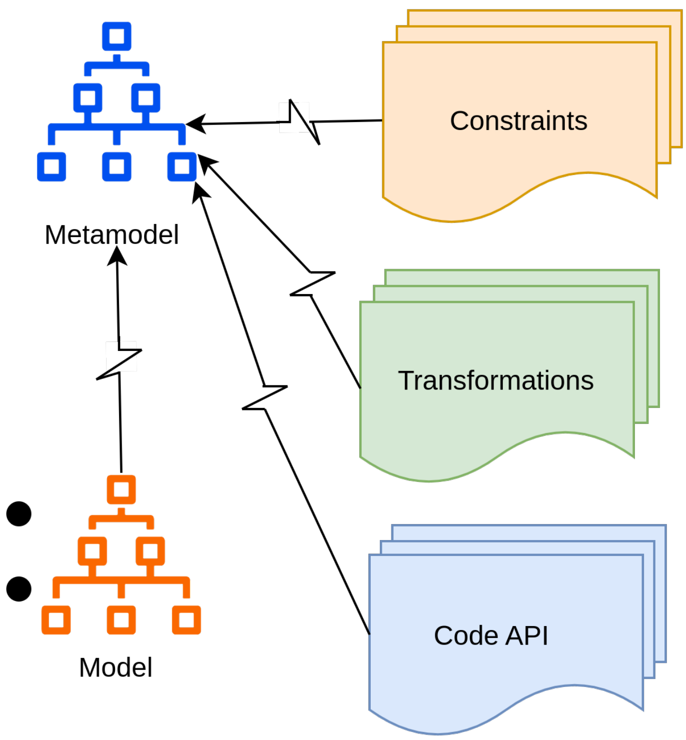

Figure 2.2 – MDE Ecosystem (not sure to include)

Chapter 3

State Of The Art

In this chapter, I present an overview of what has been done in the field of code co-evolution in the context of Model-Driven Engineering. 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.

3.1 Metamodel change detection

One of the intrinsic properties of software artifacts is its continuous evolution [40]. Like any software artifact, metamodels are meant to evolve to meet the represented domain. In this thesis, our 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. [41]: automation degree, types of detected changes, considered issues (overlap, indefinite length, hidden changes, order of changes, and undo operations)[41].

Furthermore, many of them classified the detected changes based on their impact on the treated artifact (e.g., models, constraint, transformation, and code). I table **??** I put the largest set of changes types that I found in literature [42]. Later in Section 4.1, I will 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

Two types of evolution changes are considered when evolving a metamodel: *atomic* and *complex* changes [41], [43]. Atomic changes are additions, removals, and updates of a metamodel element. Complex changes consist of a sequence of atomic changes combined together [44], [45]. 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 [43]. Many approaches in the literature [44], [46]–[51] exist to detect metamodel changes between two versions.

In Model-Driven Engineering area, metamodel changes can be divided into three categories[52]:

— 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.

In API evolution context, API changes can be classified as Non-breaking API Changes or Breaking API Changes. 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. A non-breaking change is backward compatible. This kind of changes aims to extend the functionalities or fix errors [53].

All these approaches propose structural changes detection: Additions, deletions, or modifications to classes, attributes, associations, or inheritance structures. Definition of metamodel: definition of the structural side of the representation of the given domain. Behavioral side is given through constraints. Following this definition, in this section, I give the structural changes. Another usage ? behavioral impact ? structural impact? structural error ? behavioral error? static/ runtime?

Demuth et al. [54], Herrmannsdoerfer et al. [39], Khelladi et al. [55] are online approaches. These approaches take into consideration undo operations. Where other approaches [44], [47]–[49], [56]–[61] are offline. Offline detection may cause an order issue, particularly in [47], [49], [56]–[61]. All of them do not consider hidden changes except Vermolen et al. [44].

Herrmannsdoerfer et al. [39] and Williams et al. [47] are manual. Di Ruscio et al. [56], Vermolen et al. [44], and Khelladi et al. [55] are all semi-automatic that require user decision to select final output changes. Demuth et al. [54]; Levendovszky [57],Cicchetti et al. [48], Garces et al. [58], Langer et al. [49], Garcia [59], Xing et al. [60], Moghadam et al. [61] are full automatic approaches.

Moghadam [61], Vermolen [44] and [55] take into consideration indefinite length of a complex change. Khelladi et al. [55] take into consideration overlap issue.

After studying the literature change detection approaches of metamodel, I found that Khelladi et al. [55] handle all type 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 khelladi2016detecting approach is their output representation and vocabulary. It required a minimum effort of adaptation because I use the same changes representation.

The usage of Khelladi et al. [55] in my work is explained in section??

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 if the solution is applied during the evolution of the metamodel, we call it an online solution, otherwise it is offline. 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, the model becomes no conformant. A set of resolutions are applied to co-evolve the model to gain again its conformity to the metamodel. Two strategies to evolve models due to metamodel evolution. 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 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 [62]. The first category is Resolution Strategy Languages that specifies in a transformation language how to update the model given the list of metamodel changes [63]–[69]. The category Resolution Strategy Generation groups approaches that generate full or partial resolution for each metamodel change [58], [70]–[73].

The third group of "Predefined Resolution Strategies" contains approaches that provide automation, when it is possible, by applying predefined resolution strategies [74]–[82].

Some of these approaches requires 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.[83].

The fifth and last category is called Constrained Model Search. It groups approaches that do not use the metamodel change, but uses the original model and the new metamodel to apply a constrained-based search of valid model variants [84]–[86]. Other approaches consider the model co-evolution problem as an optimization one that does not need the list of changes of the metamodel [87]–[89].

Depending on whether a user intervention is need or not to apply predefined or generated resolutions, we distinguish automatic and semi-automatic approaches. Di Ruscio et al. [90],MCL (Levendovszky et al. [57], Anguel et al. [73],cicchetti et al. [91], brand et al. [79], CBRMig[83] are automatic model co-evolution approaches.

Semi-automatic approaches that I find in literature are: de Geest et al. [71], Garcès et al [92], Wittern et al. [82] (for atomic changes, the co-evolution is automatic, for complex changes, the co-evolution is manual).

COPE [39], and Wachsmuth et al. [77], Kessentini et al. [89], [93] do not explicitly study the impact of metamodel evolution on models.

Cicchetti et al. [91] 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. [58] 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 if metamodel evolution on the models.

Demuth et al.[84] consistent change propagation focuses on maintaining consistency between artifacts (metamodel itself and existing models). Impacted constraints are used to propose repairs (to regain model conformance), then manual intervention is needed to select and apply these repairs. (metamodel changes are propagated through constraints that are used to repair moodel inconsistencies (by model and possible modification of metamodel)

3.2.2 Metamodel and constraints co-evolution

Another artifact that depends on the metamodel and needs to be adapted to the evolution of its metamodel 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. [94] proposed a template-based, that cannot cover all changes types, of the predefined structure of the updated constraint. Markovich et al. [95] proposed refactoring rules that depend on the impact of UML class diagram evolution on the constraints. In this approach, the user select the refactoring rule to be applied on the model then on the depending constraints.. Hassam et al. [96] propose METAEVOL, based on a transformation language. Kusel et al. [97] propose a solution for the co-evolution of the constraint body and do not include its context that may need co-evolution also. Cabot et al [98] treats OCL constraints co-evolution due to metamodel deletion change. Khelladi et al. [99] proposed an approach that record the metamodel atomic and complex changes in a chronological order then apply one or many resolutions to co-evolve the constraints. Batot et al. [100] tackle the constraint co-evolution problem as an multi-objective optimization problem and apply heuristic-based recommendation approach that does not require/use a predefined set of transformation rules/resolutions to co-evolve the constraints.

Kusel et al. [97] studies the impact of metamodel evolution on OCL expressions. They distinguished between breaking and non-breaking impact. moreover, they divided the changes into three groups : constructives are non breaking where destructive changes, and updative changes are breaking changes. These breaking changes, are considered so if they have at least one breaking impact case. They give a table that contains every possible case, whether it is breaking or not and the corresponding resolution. This paper proposes

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.

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)[101]. a set of Pamomo specifications (input models) are verified before and after OCL expressions' co-evolution. Pamomo states. properties the models must fulfill

Batot et al. [102] register metamodel elements that were deleted, added, or had their multiplicity changed between the two versions. Computing atomic differences does; but does not require high level changes identification. They do not explicitly analyze the impact of metamodel evolution on OCL constraints. Their goal is to satisfy the objectives of NSGA II but no more checking is done. it proceed to syntactic comparison to determine if a candidate constraint is the same as the expected one.

Demuth, markovitch, cabot [94], [95], [98] their approaches ar fully automatic while Hassam, kusel and Khelladi, and Batot [96], [97], [99], [100] propose semi-automatic approaches since the user must select from the recommended output constraints.

Hassam et al [96] proceed by a partial impact analysis, through a table that contains constraint context that is linked with the involved elements of mm that is evolved, the designer is responsible to check the validity of the constraint.

Khelladi et al. [99] associate each evolved metamodel element to impacted (context and body) and precise impacted astnodes in a table. no further checking

Other approaches do not study the impact of metamodel evolution on the constraint artifact like Batot et al. [100].

Cabot et al. [98] as an impact analysis, identify of the elements to be deleted (those selected by the user plus all the elements affected by them) Cherfa et al. [103] provide an assistance to developers OCL constraint co-evolution, by focusing on the structures in the metamodel that potentially cause problems and which need new OCL constraints after the co-evolution). Their approach no explicitly process an impact analysis.

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. Mendez et al. [104], Ruscio et al. [105], Garces et al.[92], Kusel, et al.[106], and khelladi et al. [107] proposed to co-evolve impacted transformations with a set of resolutions. T Mendez et al. [104], Ruscio et al. [105], khelladi et al. [107] while they propose to evolve impacted transformation, no impact analysys is explicitly studied. Garces et al.[92]

explains the impact of the evolution of the metamodel on the transformations through a motivating example but not an explicit study inddepend from the example.they are semi-automatic approaches.

Garcia et al.[59] proposed an ATL transformation-based approach, which means that their resolutions are ATL transformations. This approach only guarantees that the transformation is syntac- tically correct, mention that any other correctness properties, inclusing semantic correctness is not taken into consideration. Kusel, et al.[106] explained explicitly that his approach verify semantic correctness by properties expressed in PaMoMo language as a kind of regression testing.

The approach of khelladi et al. [107] covers the largest set of possible resolutions whereas Ruscio et al. [105], estimate the cost of the co-evolution to decide about the co-evolution. They further explored the variability of the co-evolution due to the possible alternative resolution. All the approaches propose unique resolution to co-evolve each transformation. Ruscio et al. [105] approach allows developers to manually replace or refine a resolution. Khelladi et al. [107] in their ()change probagatoion-based approach, allow to compose existing resolutions into a new one. This approach does not process any semantic verification of the transformation co-evolution. Kessentini et al. [108] 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.(no semantic verification of the approach's correctness).

Eventhough I consider the co-evolution in larger diameter 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 by 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], [58], [77], [88], [91]–[93], [99], [102], [106]–[111].

Yu et al. [10] proposed to co-evolve the metamodels and the generated API in both directions. However, they do not co-evolve the code on top of it.Khelladi et al. [11] proposed an approach that propagates metamodel changes in the code as a co-evolution mechanism. However, 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 with no comparison to a baseline.

3.3.2 API and client code co-evolution

Existing approaches for code migration are related to our work. Henkel et al. [112] proposed 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 only the changes renames, moves, and type changes.

Nguyen et al. [113] also proposed an approach that guides developers in adapting code by learning adaptation patterns from previously migrated code. Similarly, Dagenais et al. [114]–[116] also use a recommendation mechanism of code changes by mining them from previously migrated code. Anderson et al. [117] 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. [118] extract a set of mapping rules and apply code-based transformations to update its clients.

Zaitsev et al.[119], present a survey about library evolution, that involves developers from Two Industrial Companies:Arolla and Berger-Levrault, and Pharo as an Open-Source Community. In this survey, the study was conducted in both perspectives :client side and library side. Kula et al. [120] 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 [121]–[126]. Jezek et al. [127] 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 can be found in the PhD thesis of zaytsev Oleksandr [128]. Shaikh et al. [129] 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 [130]–[132] rely on pre-collected examples to learn how to evolve the additional client code. Xu et al. [133] instead of learning from code examples, it constructs a database of edits to use during clients' migration.

Fazzini et al. [131] propose to check the code migration using differential testing but it still needs previous example-based learning to update the client code. Zhong et al. [134] proposes "LibCatch", a tool 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. They further 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. [135] DeepMig: A transformer-based approach to support coupled library and code migrations : to add

3.3.3 Automatic Program Repair

Xia et al. [136] conducted a study on the application of Pretrained language models including both generative and infilling models on APR. This study investigated the ability of PLM in generating correct patches and its performance in ranking these patches, in addition to its performance in scaling. Claire et al. [137] give a review article about Automatic Program Repair. Their paper present 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. [138] propose a co-evolutionary-based approach for APR. This means that they aim to evolves two populations simultaniously: a set of patches and a test suite. They implemented their workflow as a tool called EVOREPAIR as an extension of EVOSUITE.

Xia et al [139] 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.

In addition to migration approaches, extensive state of the art exists on program repair [137], [140]–[142]. However, they do not repair code errors, but rather bugs that are found due to failing tests (e.g., Meng at al. [143]). They could be used as a next step after o co-evolution.

Chen et al. [144] 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, which is equivalent to our matched pattern usages. 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, [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 [145]. Metamodels evolution [146] (Section 3.2), APIs evolution [53] (Section 3.3.2), grammars evolution in [147], schemas evolution [148], [149], and ontologies evolution [150].

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

DSL is 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. [151] 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 [152], [153], the presence of inter-DSL dependencies in an 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 evolution, works treated this topic from many aspects, and from different ecosystems and case studies [154]. In the one hand, there are opensource ecosystems case studies like The Graphical Modeling Framework (GMF) [5] 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 hint at the possible effects on the related language development artifacts [155].

In the other hand, there are industrial ecosystems case studies like CARM (Control Architecture Reference Model) is an industrial ecosystem for ASML [6] is the world's leading provider of complex lithography systems for the semiconductor industry [154].

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, for example, for hardware changes, the control of business needs, or the progress of scientific research. 2) Internal for bug fixing, or improving the verbosity of the language [156]. For example, the introduction of the generics since Java 5 [7] as new feature. Another example of an explicit semantic change in Python 2 an expression such as 1/2 returns 0. However, 1.0 / 2 returns 0.5. To 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 of 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

5. https://eclipse.dev/modeling/gmp/?project=gmf-runtime
6. https://www.asml.com/en
7. https://blogs.oracle.com/javamagazine/post/understanding-java-generics-part-1-principles-and-fundamentals

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. show that developers lack awareness of the often present limitations and possible incompatibilities which make code maintenance a hard task [157]. Regarding programming languages evolution, many works were centered around change-impact analysis before proposing an adaptation approach [158]–[160]

Urma et al.[156] 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, Expansion and evolution of the R programming language regarding linguistic understanding of human language [156]. Another programming language, Pharo [122] analyzed in their their empirical study the most important changes in Pharo API and their impact on different systems in Pharo ecosystem. Ochoa et al. [161] 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 the the dissertation of Raoul-Gabriel Urma [156].

3.4 Behavioral correctness of the co-evolution

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 [162] (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 investigate the approaches that check the impact of a code different type of evolution on the behavioral correctness of the code.

In automatic program repair, 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. [138] propose an approach to check the correctness of the generated repair patches. Their approach is an extension of EVOSUITE, that has two outputs. First output is the repair patches of better quality and the second output is the tests that proves the veracity of the patches.

In literature, different approaches exist for code translation from a programming language to another. Roziere et al. [163] leverage generated unit tests to filter out invalid translations and reduce the noise in the generated translations to have better candidates.

Qi et al. [164] also divide exsting 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. [165] states 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 systems could be biased [165]. 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 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 [165]. This paper proposes metrics that limit the biases when assessing APR tools. The list of the used metrics:

- Upper bound Repair Performance metric aims to clearly provide an indication of the patch generation limitations when focusing on this part of the APR system (i.e., APR systems are given with the exact bug-fixing positions obtained from the ground of truth developers' patches).
- Fault Localization Sensitiveness metric aims to assess the impact of the used fault localization on the repair performance of the APR system.
- Patch Generation Efficiency metrics aim to clarify the APR efficiency, the effort to yield a plausible/correct patch.
- Bug Diversity metrics aim at evaluating APR system performance from intrinsic attributes of bugs.
- Benchmark Overfitting metrics aim to clarify the difference of APR systems performance between in-the-lab and in-the-wild assessment settings.

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, means behavior-preserving program transformations for code quality improvement. Soares et al. [166] propose to check refactoring safety by checking errors due to non behavior-preserving transformations. Soares et al. [166] 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. [166]'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. [167] 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. [168] leverage generated unit tests to detect semantic conflicts in different merging scenarios, and differential generated unit tests generated and run on both commit the changed commit pairs. Their results show the efficiency of the generated unit tests and no conflict was wrongly detected.

Correa et al. [111] exploit regression testing to assess the correctness of OCL specifications' refactoring.

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 since 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 [169], comparison of the performances of Copilot with mutation-based code gener-

8. https://pmd.github.io/
9. https://plugins.jetbrains.com/plugin/4597-qaplug–findbugs
10. texthttps://www.dataversity.net/a-brief-history-of-large-language-models/

ation techniques [170], and the impact on productivity and the usefulness of Copilot for developers [171], [172]. Nguyen et al. [173] performed an early empirical study on the performance and understandability of Copilot generated code on 34 problems from Leetcode. Doderlein et al. [174] extended the study of Nguyen et al. [173] 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 Human Eval data set. Nathalia et al. [175] evaluated the Performance and Efficiency of ChatGPT compared to beginners and experts software engineers. Yeticstiren et al. [176] compared the code quality generated from Copilot, CodeWhisperer, and ChatGPT, showing an advantage for ChatGPT in generating correct solutions. Guo et al. [177] 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. Finally, 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. [178] 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. [179] explore how ChatGPT can be used in ambiguity resolution in Method Name Suggestion Log Summarization, Anaphora resolution, Python Type Inference, Commit Message Generation, Code Review, Duplicate Bug Report Detection, Natural Language Code Search, Vulnerability Detection, Code Clone Detection, Test Oracle Generation, Code Generation from natural language, Merge Conflict Resolution, and Code Refactoring. Sridhara et al. [179] found that Extract Method refactorings did not match with the developers' refactorings collected from Silva et al. [180], 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. [181] investigates the potential application ChatGPT as an LLM for bug detection and refactoring particular code bad smell detection. Hemberg et al. [182] explain their approach on how an algorithm, with the general algorithmic structure of an EA and evolutionary operators, can use an LLM to evolve code, how the operators are designed to formulate LLM prompts, task the LLM via the prompts, and 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 genetic algorithm.

Zhang et al.[183] aims 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 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 [184] run an empirical study on the quality of generated OCL constraints with Codex.

However, these studies also 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. [185] proposed self-augmented code generation framework based on LLMs called SelfEvolve. SelfEvolve allows generating code and keep correcting it iteratively with the LLM. 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. This synthesis focuses on distilling insights from 1) the co-evolution 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.

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- 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 most approaches use the metamodel changes in the process of adapting models, most of them do not explicitly study the impact of the metamodel evolution on the models except Hebig et al .[41].

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.[106].

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 (the number of papers?), I find that only Kusel et al. [110] used Pamomo[101] 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 Table 3.2. My current work 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 ??). 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 two existing approaches are fully automatic and all the rest are semi-automatic. Only three approaches are standalone without requiring a pre-learning phase before the co-evolution. My approach 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 **??** in my work. Finally, only Fazzini et al. [131] proposed to validate the co-evolved Android Apps with a similar methodology as in our work based on tests' execution.

Regarding different approaches that I browse about behavioral correctness of co-evolution, I find that just few works dedicated a space to check behavioral correctness [110], [111]. 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. studies also focused on the ability of LLMs to generate MDE artifacts, such as models and constraints, but not on their co-evolution.

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.

TODO: Conclude the section to introduce next contributions

Table 3.1 – Catalog of model operators

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 3.2 – Related work comparison (to position)

Approaches	Category	Approach	Automation	Requires pre-learning	Change types	Validation
Lamothe et al. [132]	Android api migration	Identifying migration patterns and rank them to select the most context-similar	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. [131]	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. [143]	Bug fix context	Identifying migration patterns and rank them to select the most context-similar	Semi-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete)	No ×
Wu et al. [186]	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. [114], [115]	Java library evolution	Recommendation approach for compilation errors' correction	Semi-automatic	Yes ✓	Deleted or deprecated methods	No ×
Henkel et al. [112]	Java library evolution	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. [116]	Java library evolution	Recommendation approach for API usage adaptation in client	Semi-automatic	Yes ✓	Any change in AST level (Insert/Move/Update/Delete)	No ×
Zhong et al. [134]	Java library evolution	Compiler-directed tool for migrating API callsite of client code	Fully-automatic	no ✓	N/a	No ×
Gerasimou et al. [118]	Other library Evolution	Code-based transformation to update client code	Semi-automatic	No ×	A set of mapping rule	No ×
Xu et al. [133]	Other library Evolution	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 **??**	No ×
Our approach	Model-centric evolution	Code co-evolution guided by Metamodel changes pattern matching	Fully-automatic	No ×	See Table **??**	Yes ✓

CHAPTER 4

TITRE DU PREMIER CHAPITRE

4.1 Change detection

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 4.1 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 4.1), 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 4.1). We take as input the evolution changes of the metamodel between the two versions of this metamodel $\boxed{1}$. Then, we parse the additional code $\boxed{2}$ to retrieve the list of errors. After that, we get to the bottom part of Figure 4.1, both the list of metamodel changes and the list of errors are used as inputs for the pattern matching step $\boxed{3}$. It analyzes the structure of the error to match it with its impacting metamodel change and decides which resolution to apply for the error co-evolution $\boxed{4}$. 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 $\boxed{5}$. In addition to the automatic co-evolution,

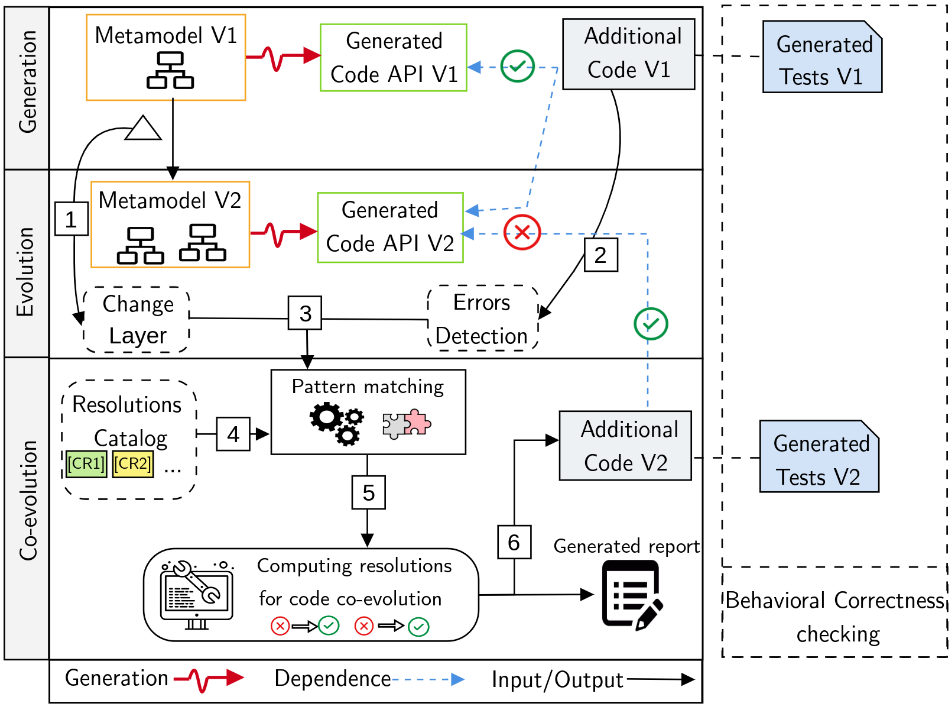

Figure 4.1 – Overall approach for metamodel and code co-evolution

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. [187] uses tests to find regressions in different versions of REST APIs. In particular, Lamothe et al. [132],test [131] 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. [143] 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.

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 cochanging. », *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. 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`.

[23] 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`.

[24] 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`.

[25] 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.

[26] 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.

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

[41] 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.

[42] 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.

[43] 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`.

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

[45] 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.

[46] 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`.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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`.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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`.

[69] 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`.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

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

[75] 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.

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

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

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

[90] 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`.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

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

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] E. Batot, W. Kessentini, H. Sahraoui, and M. Famelis, « Heuristic-based recommendation for metamodel—ocl coevolution », *in 2017 ACM/IEEE 20th Interna-*

tional Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2017, pp. 210–220.

[103] 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.

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

[105] 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.

[106] 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.

[107] 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.

[108] 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.

[109] 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.

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

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

[112] 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.

[113] 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.

[114] 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.

[115] 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`.

[116] 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`.

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

[118] 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`.

[119] 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`.

[120] 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`.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

[127] 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.

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

[129] 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`.

[130] 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`.

[131] 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`.

[132] 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`.

[133] 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`.

[134] 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`.

[135] 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`.

[136] 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`.

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

[138] 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`.

[139] 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`.

[140] 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.

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

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

[143] 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.

[144] 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`.

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

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

[147] 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`.

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

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

[150] 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.

[151] 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.

[152] 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.

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

[154] 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.

[155] 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.

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

[157] 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.

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

[159] 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.

[160] 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.

[161] 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`.

[162] 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.

[163] 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.

[164] 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.

[165] 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.

[166] 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.

[167] 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.

[168] 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.

[169] 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.

[170] 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.

[171] 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.

[172] 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.

[173] 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.

[174] 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.

[175] 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.

[176] 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.

[177] 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.

[178] 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`.

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

[180] 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.

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

[182] 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.

[183] 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.

[184] 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.

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

[186] 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.

[187] 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.

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: titre (en anglais)..............

Keywords: de 3 à 6 mots clefs

Abstract: 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.