On the Power of Abstraction: a Model-Driven Co-evolution Approach of Software Code

Djamel Eddine Khelladi, Benoit Combemale, Mathieu Acher, Olivier Barais

On the Power of Abstraction: a Model-Driven Co-evolution Approach of Software Code

Djamel Eddine Khelladi
CNRS, IRISA, Univ. Rennes
Rennes, France
djamel-eddine.khelladi@irisa.fr

Benoit Combemale
Université Toulouse & Inria Rennes
Rennes, France
benoit.combemale@irisa.fr

Mathieu Acher, Olivier Barais
Univ Rennes, Inria, IRISA
Rennes, France
(mathieu.acher,olivier.barais)@irisa.fr

ABSTRACT

Model-driven software engineering fosters abstraction through the use of models and then automation by transforming them into various artefacts, in particular to code, for example: 1) from architectural models to code, 2) from metamodels to API code (with EMF in Eclipse), 3) from entity models to front-end and back-end code in Web stack application (with JHispter), etc. In all these examples, the generated code is usually enriched by developers with additional code implementing advanced functionalities (e.g., checkers, recommenders, etc.) to build a full coherent system. When the system must evolve, so are the models to re-generate the code. As a result, the developers' enriched code may be impacted and thus need to co-evolve accordingly. Many approaches support the co-evolution of various artifacts, but not the co-evolution of code. This paper sheds light on this issue and envisions to fill this gap.

We formulate the hypothesis that the code co-evolution can be driven by the model changes by means of change propagation. To investigate this hypothesis, we implemented a prototype for the case of metamodels and their accompanying code in EMF Eclipse. As a preliminary evaluation, we considered the case of the OCL Pivot metamodel evolution and its code co-evolution in two projects from version 3.2.2 to 3.4.4. Preliminary results confirms our hypothesis that model-driven evolution changes can effectively drive the code co-evolution. On 562 impacts in two projects' code by 221 metamodel changes, our approach was able to reach the average of 89% and 92,5% respectively of precision and recall.

ACM Reference Format:
Djamel Eddine Khelladi, Benoit Combemale, and Mathieu Acher, Olivier Barais. 2020. On the Power of Abstraction: a Model-Driven Co-evolution Approach of Software Code. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

Model-Driven Engineering (MDE) has proven to be effective in the development and maintenance of large scales systems [7, 8]. MDE aims to tackle increasing software development complexity by using abstraction and automation [10]. Abstraction is achieved by creating and reasoning on various models expressing various concerns of a software system, e.g., structural, behavioral, requirements and business aspects. Automation is achieved by leveraging and transforming those models into various artifacts, such as models, scripts, documentation, source code, etc.

Many use cases have been reported in the literature where different types of models and of code co-exist. For example, *JHipster* proposes to generate, from *entity models*, modern web applications and microservices with a front-end and a back-end code. JHipster gained a lot of popularity in the last few years among developers, and more than 290 companies from all around the world adopted it, such as Google, Ericsson, HSBC, Siemens, etc[1]. Out of an entity model, numerous configuration files (e.g., Docker, pom.xml), database-specific artefacts, Java, JavaScript, HTML, and CSS code are generated. In practice, developers then face the challenge of enriching the generated code both on the client and the server side to implement new functionalities and more complex behavior.

Another example is the implementation of languages with the *Eclipse Modeling Framework (EMF)*[2]. Based on a *metamodel*, EMF supports the generation of code consisting of a core API, adapters, serialization facilities, and an editor. This code can be used to load and manipulate model instances. More importantly, it is also enriched by developers to offer additional functionalities for the software language and its tools. Today, many Eclipse projects[3] leverage on the code generated by EMF to support developers in various tasks (e.g., developing advanced editors, debuggers, checkers, etc.). As a result, many languages have been developed using EMF and its ecosystem, such as BPMN [17], OCL [18], or UML [19]. In all these examples, developers always enriched the generated core API implementing advanced functionalities and model transformations. These are two use cases of co-existence of models and code in practical settings. Many other approaches propose to also generate code skeleton from design or architectural models [2, 25, 26], which developers must enrich it as well.

One of the foremost challenges to deal with in *MDE* is the evolution of models and their impacts. As a result of a model evolution, after the originally generated code is re-generated, the additional code implemented by developers is likely to be impacted and must be co-evolved accordingly. As different kind of models with different levels of abstraction exist, code co-evolution becomes even more challenging. Hence, the few attempts in the literature [1, 9, 21, 22] to handle the code co-evolution addressed a single use case for a specific type of models and code.

This paper ambitions to address the challenge of code co-evolution with evolving models with a generic change propagation approach. Our hypothesis is that the code enriched after its generation, can be co-evolved driven by the model's evolution by means of change propagation. Thus, we leverage on the power of abstraction of models that comes with MDE for the code co-evolution. Change propagation showed to be efficient in many domains, such as in the context of artifact recommendation or co-evolution of models or transformations [4, 5, 14].

[1] Cf. https://www.jhipster.tech/companies-using-jhipster/
[2] Cf. https://www.eclipse.org/modeling/emf/
[3] Cf. https://marketplace.eclipse.org

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea
2020.

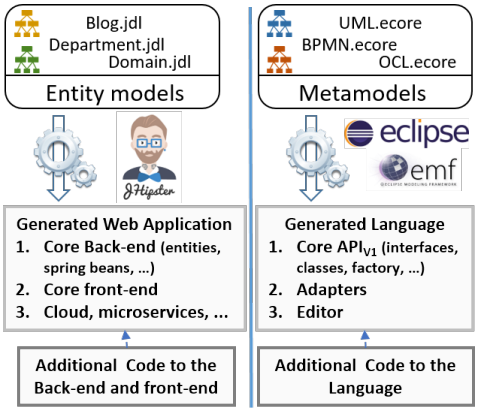

Figure 1: The JHipster and the Eclipse EMF use cases.

To investigate this hypothesis on the code co-evolution, we implemented a prototype for the EMF Eclipse use case of metamodel evolution and its accompanying code co-evolution. We run an impact analysis to identify all impacts at the code level. Then, we propose resolutions that are used as a basis for our code co-evolution in response to model evolution.

As a preliminary evaluation, we considered as a case study the implementation of the OCL language [18] in Eclipse. In particular, the case of the OCL Pivot metamodel evolution and its code co-evolution in two medium-sized projects from version 3.2.2 to 3.4.4. We evaluated to what extent our approach can correctly co-evolve the projects' code in response to the metamodels' evolution. Preliminary results confirm our hypothesis that evolution changes can effectively drive the code co-evolution. On 562 impacts in two projects' code by 221 metamodel changes, Our approach was able to reach 89% and 92,5% respectively of precision and recall.

2 MOTIVATING EXAMPLE

In this section, we illustrate the code co-evolution challenges on two cases, namely JHipster and EMF Eclipse, while highlighting similarities and differences.

JHipster is a development platform to generate, develop and deploy modern Web applications with Spring Boot, Angular/React/Vue and Spring microservices. From a specified *entity model*, JHipster generates a minimal working web application ready for deployment. Developers can then extend it with new functionalities both on the back-end and front-end sides of the web application.

The Eclipse Modeling Framework (EMF) is a framework for building tools and other applications based on a structured *metamodel*. It follows a similar process as jHipster to generate a minimal software language from the *metamodel*, consisting of an API, adapters, and a basic editor. Similarly, developers herein can then extend it and code new functionalities.

Figure 1 summarizes how JHipster and EMF work. Although these two cases address different needs for developers (e.g., different types of code and artefacts are generated), they share a common challenge. When developers must evolve their code after they have spent effort to enrich the generated code, they will evolve the metamodel in the case of Eclipse EMF and the entity model for the case of JHipster. As a result of code re-generation, all additional

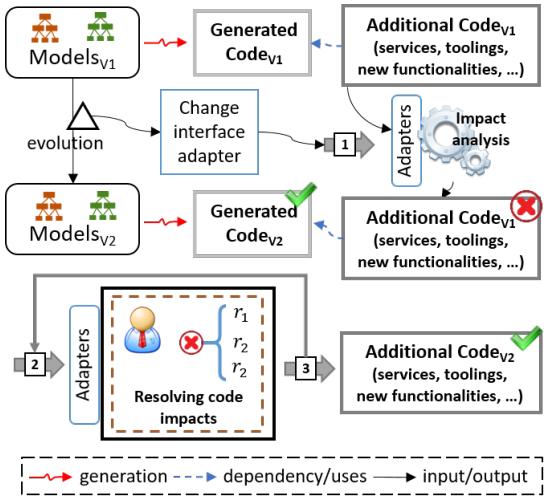

Figure 2: Overall approach.

code will likely be impacted and must be co-evolved. For instance, as Eclipse releases two versions of its IDE per year since 2018, most of the EMF Eclipse projects also release new versions. Thus, going through evolution and co-evolution frequently as well. To the best of our knowledge, unfortunately, this remains a manual task for developers, which is tedious, error-prone, and costly.

This paper aims to better support code co-evolution. Our hypothesis is that *the high level information from the applied model evolution can drive the code co-evolution.* In a sense, the "power of abstraction" can fill the gap: This is the basis of our proposed approach in this paper.

3 OVERALL APPROACH

Figure 2 gives an overview of our approach. It goes through an impact analysis phase 1, before co-evolving the impacted code 2 + 3. It relies on the abstraction of models and their changes to drive the co-evolution of the code. Our vision is to propose unified approach that can cope with different types of models and code, and thus handling many use cases in MDE. To this end, we design an adapter layer for both the impact analysis and the resolution.

3.1 Impact Analysis

This is an essential step to discover and locate at the code level the impacts of a given model evolution, i.e. applied changes. For this, model changes must be detected first . Numerous approaches exists in the literature to detect changes for various types of models, such as [3, 13, 15, 23, 24]. We define a *Change interface adapter* (see Figure 2) that serves to bridge existing change detection approaches with the changes specified by Herrmannsdoerfer et al. [6]. Thus, developers could easily interchange a given detection approach with another one that better meets their needs. The change interface considers both *atomic* (e.g., adds, deletes) and *complex* (e.g., move, split) changes.

To be able to identify the impacted code elements, we must access them. For this, we parse the code to access the Abstract Syntax Tree (AST) to access all code elements. This is part of the adapter layer that allows to fine tune it for each considered code

language. We then establish a mapping table mt between the model elements $me_1, ..., me_n \in \mathbb{M}$ and the code elements as AST nodes $ce_1, ..., ce_n \in \mathbb{C}$, i.e., $mt = Map < me_i, List < ce_i >>$. After that, for a given model change mc_i on a given me_i, we can access its impacted code elements ($List < ce_i >$) that must be co-evolved.

3.2 Change propagation-based co-evolution

The proposed approach aims to co-evolve code by means of propagating the models changes to the code. It should to be useful in our previous works with constraints [11] and transformations [14].

It first retrieves the impacted code parts by a given model change mc_i from the mapping table mt computed by our impact analysis. Then for each impact, it proposes resolutions $r_1, ..., r_n \in \mathbb{R}$ that can propagate the impacting model change. A developer can choose among the alternative resolutions which one fits her needs. This acts as a user acceptance of the resolution to be applied. Finally, the chosen resolutions are applied on the impacted code parts.

For the purpose of change propagation, we constructed a catalogue of resolutions inspired from previous work of co-evolution, and we adapted to code co-evolution. For example, due to a move property *prop* (e.g., an attribute, a method, etc.) through a reference/association *link* in the model. A call to *prop* in the code is updated to *link.prop*. Our list of resolutions is detailed in [12].

4 PRELIMINARY EVALUATION

We have instantiated our above co-evolution approach and implemented it on the use case of metamodel evolution and code co-evolution in EMF Eclipse. This section presents the preliminary results of our evaluation.

4.1 Case study and Evaluation Process

As a case study we consider the Object Constraint Language (OCL) [18] a standard language defined by the Object Management Group (OMG) to specify first-order logic constraints. In particular, we consider the OCL language implementation in Eclipse. It has been developed and maintained since 2006 and has been evolved 45 times. We considered an original version 3.2.2 and an evolved version 3.4.4 of the case study as it covers extensive changes both at the metamodels and the code levels.

In this case study, we considered the evolution of the OCL Pivot metamodel that is well documented and consists of 221 changes. We searched for projects that depend on the OCL Pivot metamodel and we found two Java projects (i.e., projects that are dependent on the metamodels' generated core API). The two projects were impacted by the OCL Pivot metamodel evolution. We collected the original and evolved Java code of those two projects.

Table 1 details the selected OCL case study in particular about the OCL Pivot metamodel and the applied changes during evolution from version 3.2.2 to version 3.4.4. Table 2 further reports the code size of the two Java projects (referred to as P1 and P2) from the original versions that must co-evolve. The goal of this preliminary evaluation is to investigate code co-evolution with evolving models. We formulate the following hypothesis:

H_{Co-Evo} The code enriched after its generation, can be co-evolved driven by the models applied changes by means of change propagation.

Table 1: Details of the Pivot metamodel evolution changes.

Atomic changes	Complex changes
Deletes: 2 classes, 16 properties, 6 super types	
Renames: 1 class, 5 properties	1 pull property
Property changes: 4 types; 2 multiplicities	2 push properties
Adds: 25 classes, 121 properties, 36 super types	

To this aim, we measure the correctness of our approach by using the two metrics *precision* and *recall* that vary from 0 to 1, i.e., 0% to 100%. They are defined as follows:

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

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

The *ProposedResolutions* are the resolutions applied by our approach while the *ExpectedResolutions* are the actual manually performed resolutions by developers.

4.2 Results

After running the impact analysis, we found the impacted code parts by the 221 metamodel changes for which resolutions were proposed by our co-evolution approach. To co-evolve all impacted parts, 562 resolutions were applied during the change propagation. This shows the applicability of our co-evolution approach that was able to handle all different impacts in the code caused by the metamodel evolution changes. To further assess the correctness of the performed co-evolution, we measure precision and recall.

For the project P1, measured precision and recall were both 92%, and for project P2, they were 86% and 93%. On average, our code co-evolution was able to reach respectively 89% and 92,5% of precision and recall. This means that our applied co-evolution propagating the impacting metamodel changes covered the expected resolutions by the developers in 89,5% and were correct in 92,5%. This confirms our hypothesis H_{Co-Evo} that propagation of models changes can effectively drive the co-evolution of code. More detailed results with additional case studies are in [12].

5 RELATED WORK

This section focuses on closest related work about model and code co-evolution. Kanakis et al. [9] showed in an empirical study that reporting to developers about model-code inconsistencies increases significantly the quality of repairs compared to when they are not reported. Further, Passos et al. [20] performed and empirical study on feature models and c code, in particular, mining variability coevolution patterns. Riedl et al. [22] proposed an approach to check the consistency between UML models and code. The goal is to identify inconsistencies that developers must repair afterward. While we focus on the direction of model to code co-evolution, they considered consistency checking in both directions. However, both do not not propose to support the repair of the code impacted parts by model changes. Similalry, Aldrich et al. [1] proposed an approach to check the consistency between architectural models with its code implementation, but without repairing them. Langhammer et al. [16] also proposes to co-evolve architectural models with

Table 2: Details of the projects and their impacts caused by the Pivot metamodel evolution.

Evolved metamodel	Projects to co-evolve in response to the evolved metamodels	N^o of packages	N^o of classes	N^o of LOC	N^o of Impacted classes	N^o of total impacts
OCL	[*P1*] ocl.examples.pivot	22	439	74002	47	532
Pivot.ecore	[*P2*] ocl.examples.xtext.base	12	181	17599	9	30

code. Pham et al. [21] further proposed an approach to synchronize architecture models (e.g., UML state machines) with generated code. However, they only focus on changes to the generated code and its structure, but not on the additional code enriching it, as in this work. Furthermore, all existing works focus on one use case of models. While we evaluated our approach's applicability on one use case of metamodels in EMF Eclipse, we aim to build on it to support other use cases of models, such as with entity models in jHipster, UML models, etc.

6 CONCLUSION

This paper aims to support developers in their activity of code co-evolution with evolving models. It leverages on the model changes and propagates them on the code. We implemented and evaluated our approach on the use case of metamodel and EMF. On two projects from the OCL language in Eclipse, results showed to be promising by applying change propagation for code co-evolution with an average of 89% and 92,5% respectively of precision and recall. In future work, we plan to extend our approach on other use cases of models-code, such as JHipster, by implementing adapters for other types of models and code languages. Then, we plan to evaluate on more case studies while diversifying their source from different kinds of models and generated artefacts.

Acknowledgments This research received funding from the CNRS PEPS, and from the AIS Rennes Metropole grant no. 190270.

REFERENCES

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* IEEE, 187–197.

[2] Franck Chauvel and Jean-Marc Jézéquel. 2005. Code generation from UML models with semantic variation points. In *International Conference on Model Driven Engineering Languages and Systems.* Springer, 54–68.

[3] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. 2009. Managing dependent changes in coupled evolution. In *Theory and Practice of Model Transformations.* Springer, 35–51.

[4] Davor Čubranić and Gail C Murphy. 2003. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th international Conference on Software Engineering.* IEEE Computer Society, 408–418.

[5] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E Lopez-Herrejon, and Alexander Egyed. 2016. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software* 111 (2016), 281–297.

[6] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. 2011. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In *Software Language Engineering*, Malloy, Staab, and Brand (Eds.). Springer, 163–182.

[7] John Hutchinson, Mark Rouncefield, and Jon Whittle. 2011. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 633–642.

[8] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 471–480.

[9] Georgios Kanakis, Djamel Eddine Khelladi, Stefan Fischer, Michael Tröls, and Alexander Egyed. 2019. An Empirical Study on the Impact of Inconsistency Feedback during Model and Code Co-changing. *Journal of Object Technology* 18, 2 (2019), 10:1–21. https://doi.org/10.5381/jot.2019.18.2.a10

[10] Oliver Kautz, Alexander Roth, and Bernhard Rumpe. 2018. Achievements, Failures, and the Future of Model-Based Software Engineering. In *The Essence of Software Engineering*, Volker Gruhn and Rüdiger Striemer (Eds.). Springer, 221–236. https://doi.org/10.1007/978-3-319-73897-0_13

[11] Djamel Eddine Khelladi, Reda Bendraou, Regina Hebig, and Marie-Pierre Gervais. 2017. A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution. *Journal of Systems and Software* 134 (2017), 242–260.

[12] Djamel Eddine Khelladi, Benoit Combemale, Mathieu Acher, Olivier Barais, and Jean-Marc Jezequel. 2020. Co-Evolving Code with Evolving Metamodels. In *2020 IEEE/ACM 42st International Conference on Software Engineering (ICSE).*

[13] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. 2016. Detecting complex changes and refactorings during (meta) model evolution. *Information Systems* (2016).

[14] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. 2018. 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, 404–414.

[15] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. 2013. A posteriori operation detection in evolving software models. *Journal of Systems and Software* 86, 2 (2013), 551–566.

[16] Michael Langhammer. 2013. Co-evolution of component-based architecture-model and object-oriented source code. In *Proceedings of the 18th international doctoral symposium on Components and architecture.* 37–42.

[17] OMG. 2015. Object Management Group. Business Process Model And Notation (BPMN). https://www.omg.org/spec/BPMN/2.0/About-BPMN/.

[18] OMG. 2015. Object Management Group. Object Constraints Language (OCL). http://www.omg.org/spec/OCL/.

[19] OMG. 2015. Object Management Group. Unified Modeling Language (UML). http://www.omg.org/spec/UML/.

[20] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of variability models and related software artifacts. *Empirical Software Engineering* 21, 4 (2016), 1744–1793.

[21] Van Cam Pham, Ansgar Radermacher, Sebastien Gerard, and Shuai Li. 2017. Bidirectional Mapping between Architecture Model and Code for Synchronization. In *2017 IEEE International Conference on Software Architecture (ICSA).* IEEE, 239–242.

[22] Markus Riedl-Ehrenleitner, Andreas Demuth, and Alexander Egyed. 2014. Towards model-and-code consistency checking. In *2014 IEEE 38th Annual Computer Software and Applications Conference.* IEEE, 85–90.

[23] Sander D Vermolen, Guido Wachsmuth, and Eelco Visser. 2012. Reconstructing complex metamodel evolution. In *Software Language Engineering.* Springer, 201–221.

[24] James R Williams, Richard F Paige, and Fiona AC Polack. 2012. Searching for model migration strategies. In *Proceedings of the 6th International Workshop on Models and Evolution.* ACM, 39–44.

[25] Dianxiang Xu, Manghui Tu, Michael Sanford, Lijo Thomas, Daniel Woodraska, and Weifeng Xu. 2012. Automated security test generation with formal threat models. *IEEE transactions on dependable and secure computing* 9, 4 (2012), 526–540.

[26] Christoforos Zolotas, Themistoklis Diamantopoulos, Kyriakos C Chatzidimitriou, and Andreas L Symeonidis. 2017. From requirements to source code: a Model-Driven Engineering approach for RESTful web services. *Automated Software Engineering* 24, 4 (2017), 791–838.