Towards Leveraging Tests to Identify Impacts of Metamodel and Code Co-evolution

Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, Olivier Barais

▶ **To cite this version:**

Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, Olivier Barais. Towards Leveraging Tests to Identify Impacts of Metamodel and Code Co-evolution. CAiSE 2023 - 35th International Conference on Advanced Information Systems Engineering, Jun 2023, Zaragoza, Spain. pp.129-137, 10.1007/978-3-031-34674-3_16 . hal-04126496

HAL Id: hal-04126496
https://inria.hal.science/hal-04126496

Submitted on 13 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Towards Leveraging Tests to Identify Impacts of Metamodel and Code Co-evolution

Zohra Kaouter Kebaili[1(✉)], Djamel Eddine Khelladi[1], Mathieu Acher[2], and Olivier Barais[3]

[1] CNRS, Univ. Rennes 1, IRISA, INRIA, Rennes, France
{zohra-kaouter.kebaili,djamel-eddine.khelladi}@irisa.fr
[2] INSA, IUF, IRISA, Inria, Rennes, France
mathieu.acher@irisa.fr
[3] Univ. Rennes 1, IRISA, INRIA, Rennes, France
olivier.barais@irisa.fr

Abstract. Models play a significant role in Model-Driven Engineering (MDE) and metamodels are commonly transformed into code. Developers intensively rely on the generated code to build language services and tooling, such as editors and views which are also tested to ensure their behavior. The metamodel evolution between releases updates the generated code, and this may impact the developers' additional, client code. Accordingly, the impacted code must be co-evolved too, but there is no guarantee of preserving its behavior correctness. This paper envisions an automatic approach for ensuring code co-evolution correctness. It first aims to trace the tests impacted by the metamodel evolution before and after the code co-evolution, and then compares them to analyze the behavior of the code. Preliminary evaluation on two implementations of OCL and Modisco Eclipse projects. Showed that we can successfully traced the impacted tests automatically by selecting 738 and 412 tests, before and after co-evolution respectively, based on 303 metamodel changes. By running these impacted tests, we observed both behaviorally correct and incorrect code co-evolution.

Keywords: Model evolution · Code co-evolution · Unit tests · Testing co-evolution

1 Introduction

Model-driven engineering (MDE) is a software engineering approach to help and support the construction and maintenance of large-scale systems [8,9,13]. MDE promotes the use of models as first-class entities during the development lifecycle of a system. Models that conform to *metamodels* are used as inputs of code generators that leverage the abstract, domain-specific concepts. For example, the JHipster project is adopted in 344 companies[1] and proposes to generate, from

[1] https://www.jhipster.tech/companies-using-jhipster/.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
C. Cabanillas and F. Perez (Eds.): CAiSE 2023, LNBIP 477, pp. 129–137, 2023.
https://doi.org/10.1007/978-3-031-34674-3_16

entity models, the different stacks of modern web applications for both client- and server-side code. Another popular example is OpenAPI[2], where many artifacts are generated from an API specification. Some low-code development platforms rely on some of the MDE principles [4], with data models and generators to raise abstraction and hide implementation-level details. *Eclipse Modeling Framework (EMF)* [27] is another prominent example. Based on a *metamodel*, EMF generates Java code API, adapters, and an editor. This generated code is further enriched by developers to offer additional functionalities and tooling, such as validation, simulation, or debugging.

One of the problems developers face in *MDE* – and in the different technologies previously mentioned – is the impact of the evolution of metamodels on its dependent artifacts. in this paper we focus on the impact on code. Indeed, when a metamodel evolves between two releases, and as the core API is re-generated again, the additional code can be impacted. As a consequence, it must be co-evolved by the developers in the next release. However, developers should spend significant effort to ensure that the code co-evolution is behaviorally correct, i.e., without altering the behavior of their impacted code. Whereas several existing approaches automate metamodels and code co-evolution [11,12,17,25,26,30,31], to the best of our knowledge, they do not focus on checking the behavioral correctness of the co-evolved code.

This paper envisions a new fully automatic approach to check the behavioral correctness of the metamodel and code co-evolution between different releases of a language, i.e., when the metamodel evolves. Our key idea is to leverage the test suites of the original and evolved versions of the metamodel. Specifically, the approach first takes as input the metamodel changes and then locates all usages of the metamodel change elements in the generated code. After that, we recursively trace the code usages until we reach the test methods. Thus, we end up matching the metamodel changes with impacted code methods and their corresponding tests. We perform this step on both original and evolved releases to check the behavioral correctness of the code before and after co-evolution.

We ran a preliminary evaluated with a prototype implementation on 2 Eclipse projects from the implementations of OCL and Modisco over several evolved versions of metamodels. As we did not find manually written tests in those projects, we generate a test suite for each release with the best available state-of-the-art tool EvoSuite [5]. Preliminary results show that we can automatically select respectively in the original and evolved releases 738 and 412 tests based on 303 metamodel changes. When running the traced tests before and after co-evolution, we could observe two cases indicating possibly both behaviorally incorrect and correct code co-evolution. Thus, helping the developer to locate the co-evolved code that must be investigated in more detail. The rest of the paper is structured as follows. Section 2 presents our envisioned approach for test tracing, while Sect. 3 describes the preliminary evaluation. Section 4 discusses the related work. Finally, Sect. 5 concludes the paper and discusses future work.

[2] https://oai.github.io/Documentation/.

2 Envisioned Approach

This section presents our proposed envisioned generic approach for tracing model evolution to tests. First, it provides an overview. Then, it describes how to handle the model changes and how to trace their impacts until the tests.

2.1 Overview

Figure 1 represents the overall approach workflow. First, we compute the difference between the two model versions (step 1). In the original version, the additional code is the impacted one, and in the evolved version, the additional code is the co-evolved one. After that, we run the impact and the test tracing analysis (step 2). Therefore, the developer can run the tests before and after the code co-evolution to check the behavioral correctness of the co-evolved code.

2.2 Detection of Model Changes

Several existing approaches allow to detect model changes between two versions, such as [3,18,19,22,28,29]. In Fig. 1, step 1, we use a change interface as input to our test tracing approach. The change interface is a specification layer for the detected changes between two versions. Therefore, any detection approach [3,15, 19,22,28,29] can be integrated by bridging its changes to our change interface and the rest of our approach can be performed independently.

Taking into account both *atomic* (*e.g.*, adds, deletes) and *complex* (*e.g.*, move, split) changes [7], the list of impacting model changes [3,10] we consider to trace the tests is as follows: *1)* Delete property p in a class `C`. *2)* Delete class `C`. *3)* Rename element e in a class `C`. *4)* Generalize property p multiplicity from a single value to multiple values in a class `C`. *5)* Move property p from class `S` to `T` through a reference *ref. 6)* Extract class of properties $p_1, ..., p_n$ from `S` to `T` through a reference *ref. 7)* Push property p from super class `Sup` to sub classes $\texttt{Sub}_1$,...,$\texttt{Sub}_n$. *8)* Inline class `S` to `T` with properties $p_1, ..., p_n$. *9)* Change property p type from `S` to `T` in a class `C`. These changes were selected after observing many versions of evolving of various types of models [20,21].

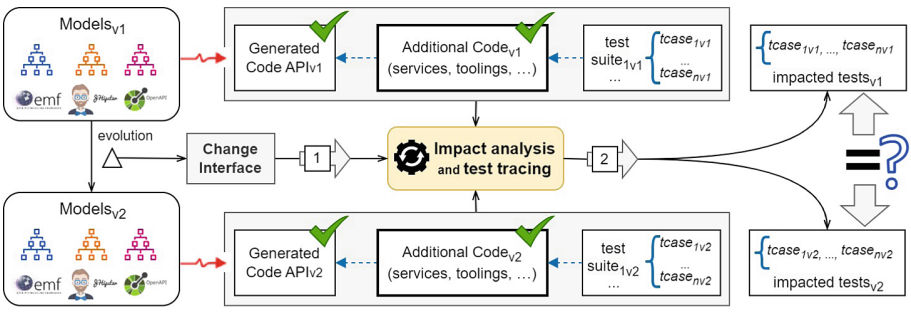

Fig. 1. Overall envisioned approach.

2.3 Tracing the Impacted Tests

Algorithm 1 describes how to check the behavioral correctness of the code co-evolution, by tracing the impact of model changes up to the test. To do that, we structure the code source to better navigate in it. Before starting Algorithm 1, we must parse the code source including tests and build the code Methods Call Graph (MCG), to retrieve the calling methods of the method invocations. Hence, trace the methods' calls recursively up to the tests. First, for each model change, we identify the list of code usages of the evolved model element. (Line 1). For example, when renaming a property called *name*, the algorithm will first find its usages, such as *getName()* or *setName()*. Then, in Line 5 we filter these impacted code usages by keeping only the ones found inside a Method Declaration, we will call this impacted method declaration *IMD*.s or if it is already treated. Therefore, after browsing all the impacted code usages, Algorithm 1 traces the list of all impacted tests, if any. For example, Listing 1.1 presents an impacted test due to a property type changing. After detecting that the type of the attribute *stereotype* is set from *Type* to *Stereotype* in the class *ElementExtension*, Algorithm 1 detects the code usage *stereotype0*. After that, it traces it to the method *test003*. As it has the *@Test* annotation, we conclude that *test003* is an impacted test due to the detected set property type. After isolating the impacted tests before and after the co-evolution, the developer can run them to investigate their results (see Sect. 3.2).

Listing 1.1. Excerpt of an impacted test in pivot project.

```
1   @Test(timeout = 4000)
2  public void test003()  throws Throwable  {
3      ...
4    Stereotype stereotype0)= elementExtensionImpl0.basicGetStereotype();
5      assertNull(stereotype0.getName());
6  }
```

3 Preliminary Evaluation

This section presents the protocol and the results of our preliminary evaluation of our prototype implementation.

3.1 Data Set and Protocol

We selected the scenario of metamodel evolution as a use case in Eclipse. In particular, we took OCL [24] and Modisco [23] projects.

Table 1 details the case studies, including their total 303 changes. Table 2 further reports the size of each version of the Java projects of both case studies. We had to generate tests using EvoSuite state-of-the-art tool [5], with the following parameters: *-DmemoryInMB = 2000 -Dcores = 4 -DtimeInMinutesPerClass = 10 evosuite:generate evosuite:export*. It uses up to 2GO of RAM, 4 CPU cores, and 10 min per test class.

Algorithm 1: Impacted tests detection

```
   Data: methodsCallGraph, change
 1 impactedUsages ← match(AST, change)
 2 impactedTests ← ϕ
 3 for (impactedUsage ∈ impactedUsages) do
 4 |   /* Find the method declaration using impactedUsage*/
 5 |   IMD ← getIMD(impactedUsage, methodsCallGraph)
 6 |   if (isTest(IMD)) then
 7 |   |   impactedTests.add(IMD) /*If not already added*/
 8 |   else
 9 |   |   parentsOfIMD ← getParents(IMD, methodsCallGraph)
10 |   |   nextRound.add(parentsOfIMD)
11 |   |   while (nextRound.hasNewIMDs()) do
12 |   |   |   IM ← nextRound.get()
13 |   |   |   if (isTest(IMD) then
14 |   |   |   |   impactedTests.add(IMD)/*If not already added*/
15 |   |   |   else
16 |   |   |   |   parentsOfIMD ← getParents(IMD, methodsCallGraph)
17 |   |   |   |   nextRound.add(parentsOfIMD)
18 |   |   end
19 end
20 return impactedTests
```

Table 1. Details of the metamodels and their evolutions.

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

3.2 Preliminary Results

With our prototype implementation, we could trace tests successfully with a total of 738 (9.4%) and 412 (7.3%) tests in the original and evolved versions, respectively, due to 303 metamodel changes. Thus, we can isolate for the developers the tests that must be executed and looked at to check the behavioral correctness of the co-evolution. Naturally, when the number of evolution changes increases, the number of tests we trace increases as well. Moreover, as several delete classes and properties occurred in the evolution changes, several tests are not generated in the evolved version, which explains why we trace more tests in the original version than in the evolved version. After tracing the tests, we could execute them to observe their effect before and after co-evolution of the

Table 2. Details of the projects and their tests.

Projects to co-evolve in response to the evolved metamodels	N^o of packages	N^o of classes	N^o of test packages	N^o of test classes	N^o of LOC	N^o of tests
$[P1_{V1}]$ ocl.examples.pivot	22	439	22	290	74002	7322
$[P1_{V2}]$ ocl.examples.pivot	22	480	22	220	89449	4990
$[P2_{V1}]$ org.eclipse.modisco.infra.discovery.benchmark	3	28	3	15	2333	524
$[P2_{V2}]$ org.eclipse.modisco.infra.discovery.benchmark	3	30	3	15	2588	619

Table 3. Selected tests before and after code co-evolution.

Projects	N^o pass	N^o fail	N^o error	Total
$[P1_{V1}]$ to $[P1_{V2}]$	106–97	2–5	347–192	455–294
$[P2_{V1}]$ to $[P2_{V2}]$	206–68	1–0	76–50	283–118

code, as shown in Table 3. In $[P1]$, even though the number of impacted tests decreased by 161, the number of passing tests decreased only by 9. Whereas the error tests decreased by 155, but, the failing tests increased by 3 from 2 to 5. This could indicate a behaviorally incorrect co-evolution. Moreover, in the other project $[P2]$, many tests that existed in the original version were not in the evolved version due to the delete changes of the metamodels, which is a sign of behavioral correct co-evolution, as the tests should be removed following the removal of the metamodel elements. These results aims to help developers to further check the code co-evolution rather than simply accepting them.

4 Related Work

This section discusses the main related work w.r.t. testing the (meta)model and the code co-evolution. Several approaches propose to automate metamodel co-evolution. Riedl et al. [26] propose an approach to detect inconsistencies between UML models and code. Pham et al. [25] propose an approach to synchronize architectural models and code with bidirectional mappings. Jongeling et al. [11, 12] propose an approach for the consistency checking between system models and their implementations by focusing on recovering the traceability links between the models and the code. Zaheri et al. [31] also propose to support the checking of the consistency-breaking updates between models and generated artifacts, including the code. Khelladi et al. [16,17] propose an approach that propagates the metamodel changes to the code as a co-evolution mechanism. However, all these approaches focus on co-evolving the code without checking the behavioral correctness of the co-evolved code. Ge et al. [6] propose to verify the correctness of refactoring with a set of condition checkers that are executed only after the refactoring application. However, our work is the first attempt that relies on a testing technique to check the behavioral correctness of the code co-evolution with the metamodel evolution.

5 Conclusion

This paper envisions an automated tracing of the impacted tests due to model evolution. By tracing the tests before and after code co-evolution, we must be able to check its behavioral correctness. We ran a preliminary evaluation on two implementations of OCL and Modisco metamodels containing generated tests with EvoSuite. Preliminary results show that we could traced 738 and 412 impacted tests based on the 303 metamodel changes. When running the traced tests before and after co-evolution, we could observe possibly both behaviorally incorrect and correct code co-evolution. This can help the developers to locate the suspicious co-evolved code. As future work, we plan to evaluate our approach on more Eclipse projects and on other case studies, such as to JHipster and OpenAPI which both generate code from a model specification similar to a metamodel. In future, we could also complement tests with formal verification, such as in [2,14] and build analyzing the build results or failures [1] before and after co-evolution.

Acknowledgement. The research leading to these results has received funding from the *ANR* agency under grant *ANR JCJC MC-EVO*^{2} *204687*.

References

1. Acher, M., Martin, H., Pereira, J.A., Blouin, A., Khelladi, D.E., Jézéquel, J.M.: Learning from thousands of build failures of Linux kernel configurations. Technical report (2019)
2. Chong, N., et al.: Code-level model checking in the software development workflow. In: The 42nd ICSE: SEIP, pp. 11–20 (2020)
3. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 35–51. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02408-5_4
4. Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M.: Low-code development and model-driven engineering: two sides of the same coin? Softw. Syst. Model. **21**(2), 437–446 (2022). https://doi.org/10.1007/s10270-021-00970-2
5. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 416–419 (2011)
6. Ge, X., Murphy-Hill, E.: Manual refactoring changes with automated refactoring validation. In: Proceedings of the 36th International Conference on Software Engineering, pp. 1095–1105 (2014)
7. Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 163–182. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19440-5_10
8. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 633–642. ACM (2011)

9. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 471–480. ACM (2011)
10. Iovino, L., Pierantonio, A., Malavolta, I.: On the impact significance of metamodel evolution in MDE. J. Object Technol. **11**(3), 3:1–3:33 (2012)
11. Jongeling, R., Fredriksson, J., Ciccozzi, F., Carlson, J., Cicchetti, A.: Structural consistency between a system model and its implementation: a design science study in industry. In: ECMFA (2022)
12. Jongeling, R., Fredriksson, J., Ciccozzi, F., Cicchetti, A., Carlson, J.: Towards consistency checking between a system model and its implementation. In: Babur, Ö., Denil, J., Vogel-Heuser, B. (eds.) ICSMM 2020. CCIS, vol. 1262, pp. 30–39. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58167-1_3
13. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47884-1_16
14. Khelladi, D.E., Bendraou, R., Baarir, S., Laurent, Y., Gervais, M.P.: A framework to formally verify conformance of a software process to a software method. In: The 30th Symposium on Applied Computing, pp. 1518–1525. ACM (2015)
15. Khelladi, D.E., Bendraou, R., Gervais, M.P.: AD-ROOM: a tool for automatic detection of refactorings in object-oriented models. In: ICSE Companion, pp. 617–620. ACM (2016)
16. Khelladi, D.E., Combemale, B., Acher, M., Barais, O.: On the power of abstraction: a model-driven co-evolution approach of software code. In: 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) (2020)
17. Khelladi, D.E., Combemale, B., Acher, M., Barais, O., Jézéquel, J.M.: Co-evolving code with evolving metamodels. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE 2020, pp. 1496–1508 (2020)
18. Khelladi, D.E., Hebig, R., Bendraou, R., Robin, J., Gervais, M.-P.: Detecting complex changes during metamodel evolution. In: Zdravkovic, J., Kirikova, M., Johannesson, P. (eds.) CAiSE 2015. LNCS, vol. 9097, pp. 263–278. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19069-3_17
19. Khelladi, D.E., Hebig, R., Bendraou, R., Robin, J., Gervais, M.P.: Detecting complex changes and refactorings during (meta) model evolution. Inf. Syst. **62**, 220–241 (2016). https://doi.org/10.1016/j.is.2016.05.002
20. Khelladi, D.E., Kretschmer, R., Egyed, A.: Change propagation-based and composition-based co-evolution of transformations with evolving metamodels. In: Model Driven Engineering Languages and Systems, pp. 404–414. ACM (2018)
21. Kretschmer, R., Khelladi, D.E., Lopez-Herrejon, R.E., Egyed, A.: Consistent change propagation within models. Softw. Syst. Model. **20**, 539–555 (2021). https://doi.org/10.1007/s10270-020-00823-4
22. Langer, P., et al.: A posteriori operation detection in evolving software models. J. Syst. Softw. **86**(2), 551–566 (2013)
23. MDT: Model development tools. MoDisco (2015). http://www.eclipse.org/modeling/mdt/?project=modisco
24. MDT: Model development tools. object constraints language (OCL) (2015). http://www.eclipse.org/modeling/mdt/?project=ocl
25. Pham, V.C., Radermacher, A., Gerard, S., Li, S.: Bidirectional mapping between architecture model and code for synchronization. In: International Conference on Software Architecture (ICSA), pp. 239–242. IEEE (2017)

26. Riedl-Ehrenleitner, M., Demuth, A., Egyed, A.: Towards model-and-code consistency checking. In: COMPSAC, pp. 85–90. IEEE (2014)
27. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
28. Vermolen, S.D., Wachsmuth, G., Visser, E.: Reconstructing complex metamodel evolution. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 201–221. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28830-2_11
29. Williams, J.R., Paige, R.F., Polack, F.A.: Searching for model migration strategies. In: Workshop on Models and Evolution, at MODELS, pp. 39–44. ACM (2012)
30. Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 540–550. IEEE (2012)
31. Zaheri, M., Famelis, M., Syriani, E.: Towards checking consistency-breaking updates between models and generated artifacts. In: Model Driven Engineering Languages and Systems - Companion (MODELS-C), pp. 400–409. IEEE (2021)