Co-evolution of Metamodels and Models through Consistent Change Propagation

Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed

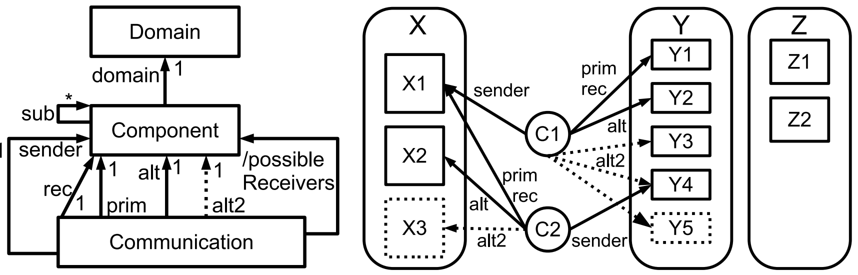
Institute for Systems Engineering and Automation
Johannes Kepler University (JKU)
Linz, Austria
{andreas.demuth|roberto.lopez|alexander.egyed}@jku.at

Abstract. In Model-Driven Engineering (MDE), metamodels and domain-specific languages are key artifacts as they are used to define syntax and semantics of domain models. However, metamodels are evolving over time, requiring existing domain models to be co-evolved. Though approaches have been proposed for performing such co-evolution automatically, those approaches typically support only specific metamodel changes. In this paper, we present a vision of co-evolution between metamodels and models through consistent change propagation. The approach addressed co-evolution issues without being limited to specific metamodels or evolution scenarios. It relies on incremental management of metamodel-based constraints that are used to detect co-evolution failures (i.e., inconsistencies between metamodel and model). After failure detection, the approach automatically generates suggestions for correction (i.e., repairs for inconsistencies). Preliminary validation results are promising as they indicate that the approach computes correct suggestions for model adaptations, and that it scales and can be applied live without interrupting tool users.

1 Introduction

In *Model-Driven Development (MDD)* [1], metamodels are key artifacts that represent real-world domains. Therefore, they define the language of models; that is, the different elements available for modeling along with their interdependencies. Metamodels thus impose structural and semantical constraints on models [2]. Although metamodels are often perceived as static artifacts that do not change, it has been shown that the opposite is the case: metamodels do evolve over time for various reasons. For instance, there is a trend for flexible design tools with adaptable metamodels that can be tailored to different domains (e.g., [3]). Another common source for metamodel evolution are refactorings that focus on improving a metamodel's structure and usability.

Co-evolution of models denotes the process of adapting models as a consequence of metamodel evolution [4, 5]. This is a non trivial process, and incorrect co-evolution may cause models to no longer comply with their metamodels. Several incremental approaches have been proposed to support this process (e.g., [6]). Unfortunately, proposed solutions are typically limited to specific



(a) Metamodel. (b) Model.

Fig. 1. Simple metamodel (a) and model (b). Metamodel evolution and valid model co-adaptations drawn dotted.

metamodels or do not fully support all kinds of possible changes (e.g., restriction of metaproperty) [7]. In particular, existing generic approaches do not take into account domain-specific model constraints. Therefore, co-evolution of metamodels and models remains an open issue.

In this paper, we outline a generic approach that does not try to automate co-evolution in general, but that detects co-evolution failures and suggests model adaptations to co-evolve a model correctly. In particular, the approach relies on incremental constraint management that allows for efficient detection of co-evolution failures (including the absence of co-evolution). If such failures are detected, the resulting inconsistencies between metamodel and model – along with other design constraints imposed on the model – are used for finding suitable model adaptations (repairs) that establish compliance of the model with the updated metamodel and thus lead to correct co-evolution.

2 Motivating Example

To illustrate our work, we use a simple metamodel for component-oriented systems with high availability requirements, as shown in Fig. 1(a). Note that, for now, we only consider elements drawn solid – dotted elements indicate evolution which we will discuss later. The metamodel defines three classes: `Component`, `Domain`, and `Communication`. Components can have an arbitrary number of `sub`-components and must belong to a `Domain`. Domains include components that are responsible for fulfilling a common task in the system. Communications occur between a `sender` and a receiver (`rec`) component. To increase the chance of a successful communication, different components can be specified as primary (`prim`) and as alternative (`alt`) target of a communication. All possible targets of a communication are aggregated by the derived reference `possibleReceivers`. Note that the defined reference cardinalities (e.g., `1` for `prim`) implicitly define model constraints. For example, an instance of `Communication` must reference exactly one `Component` via `prim`. However, to ensure that only intended models can be built, the metamodel has been extended with the following three explicit constraints:

R1 All possible receivers of a `Communication` must be located within a single `Domain` (i.e., a set of components with a common purpose).
R2 A communication may only occur between components of different component domains.
R3 It is not permitted that a single component is used as primary and alternative target.

Note that we omit other possible domain-specific constraints as well as implicit syntactical constraints for simplicity reasons.

In Fig. 1(b), a model that complies with the metamodel is depicted. Again, ignore dotted elements for now as they indicate possible evolution which we will discuss later. The derived reference `possibleReceivers` is omitted for readability reasons. The model contains three domains: X, Y, and Z. Domain X consists of only two components ($X1$ and $X2$), Y consists of four components in total ($Y1 - Y4$), and Z consists of two components ($Z1$ and $Z2$). The model also contains two communications ($C1$ and $C2$), drawn as circles.

Let us now consider a simple metamodel evolution. To increase the availability of systems and reduce the chance of communication failures, a second alternative communication target (called `alt2`) is added to the metamodel, as indicated by the dotted arrow in Fig. 1(a). Intuitively, this metamodel evolution requires the model in Fig 1(b) to be co-evolved as a new mandatory reference was added to the class `Communication` that is instantiated twice in the model. While existing approaches can typically find model adaptations that produce a syntactically correct model – for example, by adding a new reference `alt2`, which points to an arbitrary `Component`, to every `Communication` – such adaptations may easily lead to a semantically incorrect model. In the next section, we will show how our approach handles this scenario and automatically provides user guidance that helps designers to easily find valid adaptions.

3 Co-evolution through Consistent Change Propagation

To address the issues discussed above, we propose to perform co-evolution through consistent change propagation. The approach consists of two phases:

1. Detect co-evolution failures.
2. Derive options for correction of failures.

In Phase 1, the approach detects locations where co-evolution is not performed correctly. This, of course, includes the situation of plainly missing co-evolution. In Phase 2, options for a correct propagation of the metamodel change to the affected model are derived. We will now discuss those phases in more detail and also show how the approach is applied to the evolution scenario presented above.

3.1 Phase 1: Co-evolution Failure Detection

Although metamodel evolution is likely to require model adaptations, this is not a necessity – a metamodel may also change in ways that do not affect the validity of existing models. For example, when an optional reference is added. Additionally, models may be changed manually by designers or automatically by tools after a metamodel evolution occurred, trying to co-evolve the model. Therefore, it is necessary after a metamodel change – and subsequent model adaptations – to determine whether an affected model is consistent with the updated metamodel. If it is, co-evolution was performed correctly and no further intervention is required. If, however, the model is inconsistent with the updated metamodel, co-evolution failed and additional model adaptations are necessary.

Constraint Management As we have shown in the running example, constraints can be used for ensuring both syntax and semantics. Therefore, when the metamodel evolves, constraints of both kinds may be affected. By using an incremental constraint management approach, it is possible to update constraints after metamodel changes – ensuring that models are always validated with constraints that are up-to-date.

In our example, the addition of `alt2` in Fig. 1(a) requires a new syntactical constraint to be enforced on models:

R4 Each `Communication` must reference a `Component` via `alt2`.

Consistency Checking After updating the set of constraints imposed by the metamodel, standard consistency checkings mechanisms (e.g., [8, 9]) can be used to detect inconsistencies. While the typical scenario is that an unchanged model becomes inconsistent after metamodel evolution, it is also possible that a previously inconsistent model becomes consistent without any adaptations. Additionally, model adaptations that are performed for the purpose of co-evolution may be incorrect and actually introduce new inconsistencies. If a consistency checker that uses an up-to-date set of constraints detects inconsistencies after a metamodel evolution, model adaptations are required and co-evolution was not done correctly.

After adding the new syntactical constraint defined above, any standard consistency checker will find that the model in Fig. 1(b) contains two inconsistencies: neither $C1$ nor $C2$ provide a second alternative target. Thus, our intuitive assumption that additional model adaptations are necessary for correct co-evolution has been confirmed.

3.2 Phase 2: Co-evolution Correction

Once co-evolution failures have been detected, our approach reaches Phase 2 in which those failures are corrected.

Repair Options To correct co-evolution failures and propagate the metamodel change correctly, it is necessary to find model adaptations (i.e., repair options) that transform the inconsistent model into a consistent one. Unfortunately, finding suitable adaptations is non trivial as every change to a model may not only eliminate the violation of a constraint, but it may also cause other constraints to be violated. However, single changes can of course also remove several inconsistencies at once. Due to those side effects, finding suitable corrections is a complex task that should not be performed in an ad-hoc manner. Our approach employs a reasoning mechanism that takes into consideration all design constraints present in a model to find suitable adaptations [10]. Note that using not only those constraints that are actually based on the metamodel but all design constraints, repairs can be computed with higher precision as more information is available for the reasoning engine and side effects can be computed.

Let us come back to our example. During Phase 1, two inconsistencies caused by the elements $C1$ and $C2$ were detected in the model. First, we consider the inconsistency involving $C1$. To correct the syntax and remove the violation of constraint $R4$, a reference $alt2$ to any component is sufficient. Moreover, in each domain a new component could be created and used as second alternative target for $C1$. Of course, it would also be possible to create a new component in an entirely new domain. Therefore, there are 12 options available in total: one for each of the eight existing (i.e., solid drawn) components in Fig. 1(b), one for each of the three domains, and one for a new domain with a new component. However, by also taking into account the domain-specific constraints $R1 - R3$ from Section 2, our approach computes side effects for each of those options. Due to constraint $R2$, adding either $X1$ or $X2$ as second alternative target to $C1$ is not a valid adaptation as this would violate $R2$. Additionally, the existing references `prim` and `alt` from $C1$ to components of domain Y disallow the use of any components that belong to a domain other than Y, according to constraint $R1$. This rules out any remaining options that involve a second alternative receiver in domain Z or in a newly created domain. Finally, constraint $R3$ disallows $Y1$ and $Y2$ as options because they are already possible receivers. Note that this means a reduction from 12 options – from which 9 are actually semantically incorrect – to only 3 options that co-evolve $C1$ correctly. Those are drawn dotted in Fig. 1(b).

For the communication $C2$, the constraints $R1 - R4$ can only be satisfied by adding a new component to domain X that is used as second alternative receiver, as indicated by the dotted drawn component $X3$ in Fig. 1(b).

Change Execution Although each derived repair option fixes a model, some of them may seem more intuitive and more logical to stakeholders than others. Therefore, stakeholders should choose manually which of the available repair options should be executed. However, repair options could of course be selected and executed automatically if model characteristics such as readability are of low importance.

In our example, the co-evolution of $C2$ can be done automatically as there is only one repair option. To repair the inconsistency of $C1$, a user has to decide between only three options that propagate the metamodel change correctly to the model.

4 Discussion

Let us now briefly discuss the planned implementation of our approach and preliminary validation results.

4.1 Prototype Implementation

The individual parts of the approach have been implemented in previous work. For the constraint management part, we have implemented a template-based transformation engine that generates and updates metamodel-based model constraints [11]. For the consistency checking, we rely on the Model/Analyzer consistency checking framework [12] that allows for efficient incremental addition and removal of models constraints. Finally, for the repair option generation, we have implemented a generic inconsistency repair mechanism that builds upon the Model/Analyzer framework [10].

4.2 Preliminary Performance Results

We have demonstrated in [13] that constraint management through transformation is efficient and that constraints are updated within milliseconds after a metamodel change. In [12] and [9], we have shown that the Model/Analyzer is capable of validating constraints instantly, even for large industrial models of over 100,000 model elements. Moreover, it was demonstrated that adding constraints (or removing them) is handled efficiently. For repairing detected inconsistencies, we have observed that for typical UML models less than 10 suggestions were derived, also within milliseconds [10]. Moreover, we have previously found that by considering side-effects between different constraints, the number of suggestions can be reduced even further [14].

4.3 Applicability

We have illustrated how our approach updates constraints and derives options for correcting co-evolution failures. Although we have used the proposed solution in isolation to keep the example simple and focused, it is compatible with existing automatic co-evolution techniques. When used in isolation, our approach detects the absence of necessary model adaptations as co-evolution failures. When combined with other approaches, it also detects co-evolution failures that are based on incorrect model adaptations. Therefore, our solution is not a substitute but a complement to existing technologies.

5 Related Work

Let us now discuss how the presented approach relates to other work that has been done in the field of co-evolving metamodels and models. The necessity of support for efficient and automatic co-evolution of metamodel and models was identified as a major challenge in software evolution by Mens et al. [4], and various approaches have been published to address it. Instead of seeing a metamodel evolution step as a single, complex, and manually performed change that is performed in an ad-hoc manner, Wachsmuth [15] describes metamodel evolution as a series of transformational adaptations performed stepwise. Metamodel changes are traced and qualified based on properties such as semantics- or instance-preservation. Co-transformations for models can be generated based on transformation patterns that are instantiated with the performed metamodel transformations. Cicchetti et al. [16] similarly classify metamodel and model changes. They identified dependencies between different kinds of modifications and propose an automated approach that leverages these dependencies for performing co-evolution automatically. Herrmannsdoerfer et al. [17] investigated to which degree different metamodel adaptations can be handled automatically. Note that those approaches focus on decomposing metamodel adaptations into atomic steps that are used for finding suitable co-adaptations of models. While our approach also relies on atomic metamodel modifications, we use those modifications for updating the conditions that must hold in a valid model. Our approach in general does not try to automate co-evolution of metamodels and models. Instead, the fully automated co-evolution of metamodels and constraints allows our reasoning engine to provide tool users with specific guidance on how co-evolution can be performed. Note, however, that in some cases models may also be adapted automatically (e.g., if only a single repair option exists).

Wimmer et al. [18] merge different metamodel versions to a unified metamodel and then apply co-evolution rules to models. New metaclasses are instantiated and existing model elements that are no longer required are removed. Due to issues regarding typecasts and instantiation, their co-evolution rules had to be adapted. The components used in our prototype implementation are capable of handling arbitrary metamodel adaptations, including type changes.

6 Conclusion and Future Work

In this vision paper, we have presented the outline of a novel approach for supporting the co-evolution of metamodels and models. Our approach is generic and relies on the detection of inconsistencies that occur after metamodel evolution. Those inconsistencies serve as input for a reasoning mechanism that provides as output a set of possible model adaptations for repairing – that is, co-evolving – an affected model.

The preliminary validation results are promising and suggest that the presented approach is feasible and that it can be implemented efficiently. However, these were observed in tests that were performed with prototype implementations for the individual components involved in the approach. For a complete validation, we have yet to conduct case studies with industrial models and a complete implementation that fully integrates the prototypes of individual components.

References

1. D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.
2. R. B. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *FOSE*, pp. 37–54, 2007.
3. E.-J. Manders, G. Biswas, N. Mahadevan, and G. Karsai, "Component-oriented modeling of hybrid dynamic systems using the generic modeling environment," in *MBD/MOMPES*, pp. 159–168, 2006.
4. T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *IWPSE*, pp. 13–22, 2005.
5. L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Enhanced automation for managing model and metamodel inconsistency," in *ASE*, pp. 545–549, 2009.
6. M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "COPE - automating coupled evolution of metamodels and models," in *ECOOP*, pp. 52–76, 2009.
7. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *EDOC*, pp. 222–231, 2008.
8. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service," *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 151–185, 2002.
9. I. Groher, A. Reder, and A. Egyed, "Incremental consistency checking of dynamic constraints," in *FASE*, pp. 203–217, 2010.
10. A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *ASE*, pp. 220–229, 2012.
11. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Supporting the co-evolution of metamodels and constraints through incremental constraint management," in *MoDELS*, 2013. Accepted for publication.
12. A. Reder and A. Egyed, "Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML," in *ASE*, pp. 347–348, 2010.
13. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Constraint-driven modeling through transformation," *Software and System Modeling*, 2013. DOI: 10.1007/s10270-013-0363-3.
14. A. Nöhrer, A. Reder, and A. Egyed, "Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution: NIER track," in *ICSE*, pp. 864–867, 2011.
15. G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*, pp. 600–624, 2007.
16. A. Cicchetti, D. D. Ruscio, and A. Pierantonio, "Managing dependent changes in coupled evolution," in *ICMT*, pp. 35–51, 2009.
17. M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "Automatability of coupled evolution of metamodels and models in practice," in *MoDELS*, pp. 645–659, 2008.
18. M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel, "On using inplace transformations for model co-evolution," in *MtATL*, INRIA & Ecole des Mines de Nantes, 2010.