ACM/IEEE 19th International Conference on
Model Driven Engineering Languages and Systems

October 2-7, 2016 • Saint-Malo (France)

ME 2016 – Models and Evolution Workshop Proceedings

Tanja Mayerhofer, Alfonso Pierantonio, Bernhard Schätz, Dalila Tamzalit (Eds.)

Editors:

Tanja Mayerhofer
TU Wien (Austria)

Alfonso Pierantonio
University of L'Aquila (Italy) and Mälardalen University (Sweden)

Bernhard Schätz
fortiss GmbH (Germany)

Dalila Tamzalit
University of Nantes (France)

Organizers

Tanja Mayerhofer	TU Wien (Austria)
Alfonso Pierantonio	University of L'Aquila (Italy) and Mälardalen University (Sweden)
Bernhard Schätz	fortiss GmbH (Germany)
Dalila Tamzalit	University of Nantes (France)

Program Committee

Vasilios Andrikopoulos	University of Stuttgart (Germany)
Alessandra Bagnato	Softeam (France)
Mireille Blay-Fornarino	Université Nice Sophia Antipolis (France)
Gaël Blondelle	Eclipse Foundation (Canada)
Francis Bordeleau	Ericsson (Canada)
Jordi Cabot	ICREA (Spain)
Antonio Cicchetti	Mälardalen University (Sweden)
Juan de Lara	Universidad Autonoma Madrid (Spain)
Davide Di Ruscio	Università degli Studi dell'Aquila (Italy)
Anne Etien	LIFL, University of Lille 1 (France)
Jesus Garcia-Molina	Universidad de Murcia (Spain)
Jeff Gray	University of Alabama (USA)
Yann-Gaël Guéhéneuc	École Polytechnique de Montréal, (Canada)
Rich Hilliard	MIT (USA)
Ludovico Iovino	Gran Sasso Science Institute (Italy)
Udo Kelter	University of Siegen (Germany)
Olivier Le Goaer	LIUPPA, Université de Pau et des Pays de l'Adour (France)
Tihamer Levendovszky	Vanderbilt University (USA)
Shahar Maoz	Tel Aviv University (Israel)
Tom Mens	University of Mons (Belgium)
Richard Paige	University of York (UK)
Charles Rivet	Zeligsoft (Canada)
Bernhard Rumpe	RWTH Aachen University (Germany)
Martina Seidl	Johannes Kepler University Linz (Austria)
Eike Stepper	ES-Computersysteme (Germany)
Massimo Tisi	Inria, Mines Nantes, LINA (France)
Juha-Pekka Tolvanen	Metacase (Finland)
Antonio Vallecillo	Universidad de Málaga (Spain)
Stefan Wagner	University of Stuttgart (Germany)
Konrad Wieland	Sparx Systems (Austria)
Manuel Wimmer	TU Wien (Austria)

Table of Contents

Preface

The Models and Evolution (ME) 2016 workshop addressed the evolution of artefacts of the modelling process, as inspired by analogous evolution required by software artefacts, with input from academic as well as industrial practice.

With the increasing use of model-based development in many domains, such as automotive software engineering and business process engineering, models are starting to become core artefacts of modern software engineering processes. By raising the level of abstraction and using concepts closer to the problem and application domain rather than the solution and technical domain, models become core assets and reusable intellectual property, being worth the effort of maintaining and evolving them. Therefore, models increasingly experience the same issues as traditional software artefacts, i.e., being subject to many kinds of changes, which range from rapidly evolving platforms to the evolution of the functionality provided by the applications developed. These modifications include changes at all levels, from requirements through architecture and design, to executable models, documentation and test suites. They typically affect various kinds of models including data models, behavioural models, domain models, source code models, goal models, etc. Coping with and managing the changes that accompany the evolution of software assets is therefore an essential aspect of software engineering as a discipline.

The tenth edition of the Models and Evolution workshop was co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems and represented a forum for practitioners and researchers working on the topic of evolution in modeling. We received ten papers out of which nine papers (five short papers, four long papers) were selected for inclusion in the proceedings. The accepted papers cover many different aspects of evolution in modelling including, but not limited to

- industrial practices and ecosystems,
- model evolution and co-evolution,
- model comparison,
- model synchronization,
- model change recommendation,
- model slicing, and
- anti-patterns in models.

The Models and Evolution workshop is existing in different forms since 2007. Before 2010 it was known as MoDSE and MCCM. Each edition received high attention and enough submissions for concluding that this is and remains a current and relevant topic in the theory and practice of model-based development. Thus, we would like to thank the authors—without them the workshop simply would not exist—and the program committee for their hard work.

September 2016

Tanja Mayerhofer, Alfonso Pierantonio,
Bernhard Schätz, and Dalila Tamzalit

Version Control for Models: From Research to Industry and Back Again

[Keynote]

Philip Langer
EclipseSource Services GmbH
Vienna, Austria
planger@eclipsesource.com

Keywords

model version control; collaborative modeling; model diff & merge; open-source modeling tools

1. ABSTRACT

Version control for models, including model diff & merge, is not only a crucial prerequisite for a wide-spread adoption of model-based engineering in industry, it also is and has been a popular and very active research topic since more than ten years. Several important algorithms and approaches emerged in the past to support the identification of differences among model versions, as well as to merge them into a new version. Many of those ideas have also been successfully transferred and implemented in proprietary and open-source modeling tools used in industry.

During the last few years, especially Eclipse-based open-source modeling tools, including support for version control built on Papyrus, EMF Compare, and EGit, gained significant attention and evolved into an industry-ready platform for building modeling tools in several domains. With this increasing industrial usage, however, the involved open-source technologies are challenged in different aspects ranging from the requirement for full customizability, performance with very large models to usability supporting users with strongly varying backgrounds.

In this talk, we report on our experiences gained from moving from research in the area of model diff & merge to applying and enhancing open-source technologies, such as EMF Compare, EGit, and Papyrus for industrial use. We discuss engineering and research challenges that we faced when working with industrial users and how we approach them. Several of those challenges are still open though and require a strong collaboration of researchers, technology providers, and industrial users. Open-source platforms and technologies are a great opportunity to enable such a collaboration. This is also one of the key motivations behind the Papyrus Industrial Consortium, which was founded to foster collaboration among academia and industry in the area of open-source modeling tools based on Papyrus, EMF Compare, and EGit.

2. BIOGRAPHY

Philip Langer is senior software architect and general manager of EclipseSource Services GmbH. He has more than eight years of experience in developing modeling tools based on Eclipse technologies and is an active committer on a number of EMF-based technologies. Besides, he is architecture board member of the Papyrus Industrial Consortium, where he leads the development of collaborative modeling tools to facilitate diff/merge of models within Papyrus UML, SysML, and UML-RT. Philip authored more than 50 articles in scientific journals, conferences, and workshops in the area of model-based engineering and received the Award of Excellence by the Austrian Government in 2012 for his PhD thesis on model versioning and model transformation at the Vienna University of Technology.

DSL/Model Co-Evolution in Industrial EMF-Based MDSE Ecosystems

J.G.M. Mengerink
Eindhoven University of Technology
The Netherlands
j.g.m.mengerink@tue.nl

R.R.H. Schiffelers
ASML & Eindhoven Univerisity of Technology
The Netherlands
r.r.h.schiffelers@tue.nl
ramon.schiffelers@asml.com

A. Serebrenik
Eindhoven University of Technology
The Netherlands
a.serebrenik@tue.nl

M.G.J. van den Brand
Eindhoven University of Technology
The Netherlands
m.g.j.v.d.brand@tue.nl

ABSTRACT

Model Driven Engineering and Domain Specific Languages (DSLs) are being used in industry to increase productivity, and enable novel techniques like virtual prototyping. Using DSLs, engineers can model a systems in terms of their domain, rather than encoding it in general purpose concepts, like those offered by UML. However, DSLs evolve over time, often in a non-backwards-compatible way with respect to their models. When this happens, models need to be co-evolved to remain usable. Because the number of models in an industrial setting grows so large, manual co-evolution is becoming unfeasible calling for an automated approach. Many approaches for automated co-evolution of models with respect to their DSLs exist in literature, each operating in a highly specialized context. In this paper, we present a high-level architecture that tries to capture the general process needed for automated co-evolution of models in response to DSL evolution, and assess which challenges are still open.

Keywords

Model driven engineering, evolution, domain specific language, model, maintenance, co-evolution

1. INTRODUCTION

Model driven (system) engineering MD(S)E is being used both in industry [1, 2], and open source [3] for developing software and systems. Systems design using MDE allows for analysis and feedback early in the design process. A main driver for doing so is designing domain specific languages (DSLs), *e.g.,* using the Eclipse Modeling Framework (EMF) [4]. In EMF, DSLs consist of meta-models [5] eventually augmented by OCL constraints [6]. DSLs allow one to model systems in terms of their domain, rather then using general purpose concepts offered by, *e.g.,* UML or SysML [7].

We have observed that in an industrial setting, DSLs often occur in an ecosystem (*cf.* [8, 9, 10]), *i.e.,* collection of DSLs with dependencies between them introduced,*e.g.,* by language reuse or model-to-model transformations. Additionally, infrastructural artifacts such as parsers, textual editors, and graphical editors also reside in this ecosystem.

Similarly to traditional software systems and languages [11], DSLs evolve over time [12, 13]. This leads to challenges with respect to backwards compatibility: artifacts (models, parsers etc.) from the old version of the DSL may no longer be valid in the new version of the DSL. To this extent, these artifacts have to co-evolve to reflect the changes in the DSL. This is known as the *co-evolution problem*.

Manual co-evolution of these artifacts is tedious, error-prone, and costly. To mitigate this, we wish to automate the co-evolution of artifacts in DSL ecosystems to the highest extent possible. In the literature, various approaches have been presented towards automating this co-evolution, each with their own strong and weak points [14, 15, 16, 17]. For our industrial case we aim at completeness and formality, rather than approximation of the various artifacts.

In this paper, we present *a high-level architecture that captures the various steps and components required for co-evolution of models*. Subsequently, we examine the state-of-the-art in co-evolution research to ascertain to what extent the state-of-the-art is able to effectuate the proposed architecture. Lastly, we summarize the open challenges towards implementing the architecture, and are thus future work for automating DSL/model co-evolution.

In the remainder of this paper we discuss DSL ecosystems (Section 2),and present our architecture and its components (Section 3), position existing work with respect to the architecture and determine which components are not yet supported (Section 4). Next we discuss the theoretical limitations of the automation (Section 5) and sketch the directions for future work (Section 6).

2. ECOSYSTEMS

In model-driven engineering, the meta-model is the central artifact that dictates the concepts and structure of other artifacts. Several related DSLs and their corresponding artifacts constitute an MDSE ecosystem. When evolving a DSL in an MDSE ecosystem, artifacts such as models [14, 15, 18, 19], model-to-model transformations [20, 21], text and graphical editors [22] may have to be co-evolved. Di Ruscio, Iovino and Pierantonio define three categories of co-evolving

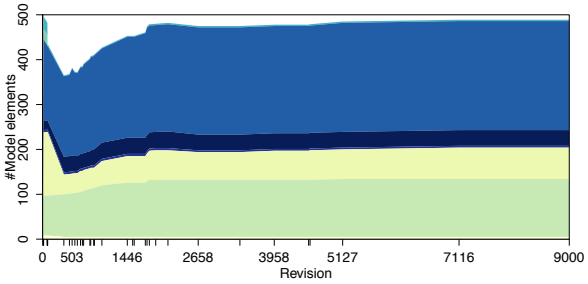


Figure 1: Evolution of the number of meta-model elements in `PGWB`. Largest groups represent references (blue), enumeration literals (indigo), attributes (yellow), classes (green). Numerous changes to the meta-model are visible.

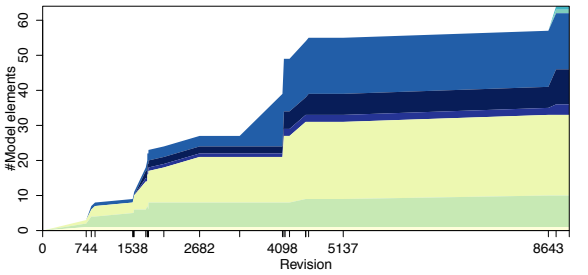


Figure 2: Evolution of the number of meta-model elements in `Basics`. Largest groups represent references (blue), enumeration literals (indigo), attributes (yellow), classes (green). Numerous changes to the meta-model are visible, in addition to a gradual increase in size.

artifacts [23]: DSL[1]/Model co-evolution, DSL/Transformation co-evolution, and DSL/Editor co-evolution.

The driving cases behind our research are several MDSE ecosystems at ASML, provider of lithography equipment for the semiconductor industry. The largest ecosystem we consider is CARM [2] consisting of 22 EMF-based DSLs, 95 QVT model transformations, and 5500 unit-test models supporting development of these transformations [24].

In the CARM ecosystem, models make up the majority of artifacts. Thus, in this work we focus on DSL/model co-evolution. The model co-evolution effort required by a DSL evolution can be expected to become bigger when carried out in the context of an MDSE ecosystem as opposed to a single DSL and its models. The reasons are threefold: reuse of concepts from other DSLs, model transformations and implicit relations between DSLs.

Indeed, meta-models may *reuse* concepts from other meta-models. However, let meta-model X reuse a concept A from meta-model Y. If A in Y evolves, models of meta-model X reusing A (from Y) might need to co-evolve. Hence, evolution of a single DSL, can cause co-evolution of models in other DSLs.

Similar ripple effects [25, 26] might be caused by *model transformations*. Assume a transformation T transforms a concept A in meta-model X to a concept B in Y. If A is evolved to include additional information, one might have to co-evolve T to incorporate this new information. Additionally, Y might not be expressive enough to encode the new information, and B itself has to be co-evolved too.

[1]We consider a DSL to consist of a meta-model enriched with OCL constraints. The original work of Di Ruscio, Iovino and Pierantonio considered meta-models only.

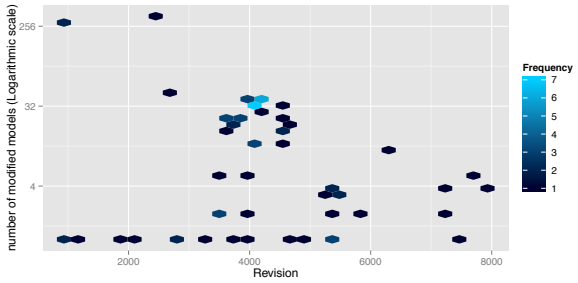


Figure 3: A binplot (aggregating several data-points into hexagonal bins) of the number of revised `PGWB` models per revision. Each hexagon represents a bin of a range of revisions and a range of "number of models modified", where the colors represent the number of data points in that bin.

Lastly, not all relations between artifacts in MDSE ecosystems are modeled (or specified) explicitly. There can also be *implicit relations*, such as the classic software co-change as described by Zimmerman *et al.* [27]. Similar relations exist for MDSE and similar solutions can be presented [28].

To illustrate the ripple effect issue in CARM consider DSLs `PGWB` and `Basics`. `PGWB` reuses concepts from `Basics`. Evolution of `PGWB` and `Basics` is shown in Figures 1 and 2, respectively. Around revision 4100 we see an increase in the number of classes, enumeration literals and references in `Basics` language. The corresponding change in `PGWB` is barely visible. However, Figure 3 shows that there is a large number of `PGWB` models that are being co-evolved around revision 4100, in response to the evolution in `Basics`.

This example illustrates that 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.

3. A HIGH-LEVEL ARCHITECTURE FOR CO-EVOLVING MODELS

As described in Section 2, several types of artifacts must co-evolve in response to DSL evolution. In this section we present a high-level architecture for co-evolving models in response to DSL evolution. However, a similar architecture may be used for other artifacts. We focus on models (rather than model transformations or editors) since models make up the majority of CARM artifacts. Moreover, we focus on co-evolving a single (arbitrary) model in response to the evolution of a single DSL. In the case where there are multiple DSLs, we can treat them as a single DSL by resolving all inclusions and dependencies. When more models are involved, the proposed process can simply be repeated for each model individually, as we have no assumptions on the model, other than that it is a valid model for the first version of the DSL.

Figure 4 presents a high-level architecture for model co-evolution. The process starts when a DSL (1) evolves (2) to

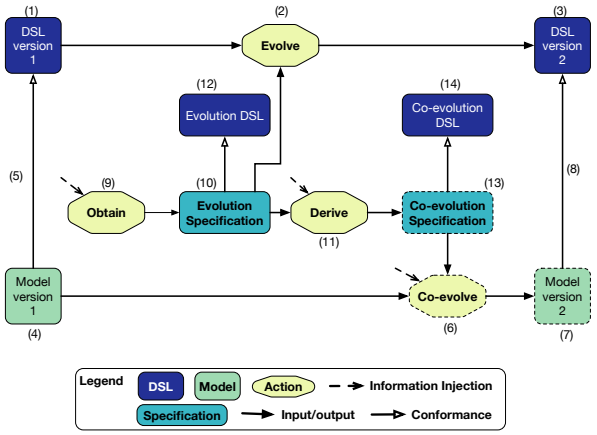


Figure 4: The framework for positioning research(questions) with respect to DSL/model co-evolution.

Table 1: A summary of the components of the high-level architecture that have to be implemented.

Comp.	Description	Fig. 4
C1	A way to model meta-model evolution	12
C2	A way to model OCL-constraint evolution	12
C3	A way to obtain an evolution specification	9,10
C4	A way to describe when a model is valid for a given DSL	1,3,4,5, 7,8
C5	A way to describe model co-evolution	13,14
C6	A way to determine if a given co-evolution specification is total	6
C7	A way to derive a co-evolution specification from an evolution specification	10,11, 13

a new version (3). Consequently, models (4) conforming (5) to that DSL (1) should co-evolve (6) into models (7) conforming (8) to the new version of the DSL (3). Since manual co-evolution of models is too costly, we aim to (partially) automate the co-evolution by providing a specification (13) of how models should co-evolve (6) in a dedicated formalism (14). To ease the creation of a co-evolution specification (13), we wish to use a DSL evolution specification (10) to derive (11) a partial co-evolution specification (13).

We stress that one can neither expect the automation nor the co-evolution specification to be *complete*. Theoretical limitations of the approach are discussed in Section 5.

To support the architecture in Figure 4, one has to implement components listed in Table 1. First of all, a formalism is required to model DSL evolution. This formalism can be further decomposed into two components: **C1**, a way to model meta-model evolution (12), and **C2**, a way to model OCL-constraint evolution (12). Furthermore, once the formalism has been chosen a separate component **C3** should focus on obtaining an evolution specification (9,10), *e.g.,* by inspecting previous changes [29].

In order to co-evolve models in response to DSL evolution, one needs **C4**, a mechanism deciding whether model (4,7) is well-formed (5,8) for a given DSL (1,3), as well as **C5** a way to describe model co-evolution (13,14).

Once **C4** and **C5** are available, one needs to determine whether all models would be correctly co-evolved by the co-evolution specification, with respect to the evolution of DSLs. As most DSLs only formally define syntax, and not semantics, this is not a question that can be answered. Hence, we merely require presence of a component, **C6**, capable of determining whether every (syntactically) valid artifact for DSL version 1 can be co-evolved with respect to a co-evolution specification to a (syntactically) valid artifact for DSL version 2. Finally, co-evolution specification depends on the evolution specification, *i.e.,* we need the component, **C7**, presenting a way to do we derive (11) a co-evolution specification (13) from an evolution specification (10).

4. EXISTING WORK

In the literature, several approaches exist that implement (part) of the architecture presented in Figure 4. As mentioned above, although approaches exist for co-evolving transformations [21, 29, 20] and editors [22], we focus on DSL/model co-evolution, as models constitute the vast majority of artifacts in the ASML ecosystems. In the remainder of this section, we discuss previous approaches to DSL/model co-evolution and map them onto our architecture. In this way we assess the state-of-the-art and identify directions for further research. Our discussion of the previous work is indebted to earlier surveys [30, 13].

The approaches we survey (primarily) target EMF-based DSLs. Additional ways to construct DSLs exist [31], with the corresponding ways of dealing with co-evolution.

Specifying Evolution: (10,12).

To allow for the specification of meta-model evolution (10), a formalism is required to capture the evolutionary steps of a meta-model from its original to its evolved version. To this extent we have computed a complete library of atomic evolutionary steps based on the meta-meta-model [32]. Using this library, every sequence of evolutionary steps can be described allowing the implementation of **C1**.

Alternatively, a generic model-to-model transformation language such as QVT [33, 34], or a meta-model independent difference DSL such as EMFCompare [35] can be used.

However, to the best of our knowledge, for OCL (**C2**) a dedicated formalism is still needed.

Obtaining an Evolution Specification: (9,10).

There are several ways of obtaining an evolution specification, which can be divided into two categories: automatic evolution specification approximation and manual evolution specification specification.

Automatic evolution specification approximation is also known as *differencing*. Such approaches such as EMFMigrate [29] and EMFCompare [35] compare the original and evolved meta-model to extract a specification of the difference. A known shortcoming of these approaches is the inability to choose between several evolution specifications that can lead from the original meta-model to the evolved one. For instance, when renaming a class, an identical result may be achieved by deleting the old class and creating a new class. Rose *et al.* [30] argue that no differencing approach can always choose the correct evolution specification.

Manual Evolution Specification Specification can be achieved by means of a predefined collection of operators, or by record-

ing changes. Operator-based methods [14] specify evolution by means of operators, that each encode a (frequently-occurring) pattern of co-evolution. The applicability of this approach relies heavily on the available operators. The state-of-the-art operator-based tool is Edapt[2] [36], which we have evaluated [37], and improved [38]. Change recording approaches record the actions performed on the meta-model by the user in order to obtain an evolution specification. This mitigates the problem presented by the differencing approach, but only is the user works in the correct way. That is, if a user deletes and adds a class, did they indeed intend a delete and add, or did they really intend a rename?

Summarizing, various approaches can be found in the literature to implement **C3**.

DSL/Model Conformance: (5,8).

When a DSL evolves, we can wonder which models are valid before the evolution, and which models are valid after the evolution.

Schoenboeck *et al.* have formalized conformance into a number of OCL constraints [39]. However, this work does not cover all constraints used by modern meta-modeling frameworks such as EMF [4]. González *et al.* have created a transformation from EMF (including OCL constraints) to a constraint solving language CSP [40]. This CSP specification then precisely describes all valid model instances. Lastly, Anastasakis *et al.* have created a similar transformation that formalizes a UML diagram [41] into Alloy [42].

Summarizing, the existing formalisms allow one to describe the set of all valid models for a given meta-model, implementing component **C4**.

Specify Co-Evolution: (13,14).

For the specification of co-evolution, several mature languages/tools exist, the most generic being QVT [33, 34]. Specifically for the co-evolution of models, a tool called Flock [43] exists. We thus consider **C5** to be adequately addressed by these existing tools.

Valid Co-Evolution Specification: (4,6,7,13).

Intuitively, a co-evolution specification takes a model in the original language as input and yields a semantically equivalent model in the evolved language as output. However, DSLs defined by a meta-model and OCL constraints merely define syntax of the models. Hence, addressing validity either calls for analysis of semantics specified elsewhere, or for redefinition of validity in terms of syntax.

Unfortunately semantics are often not formally specified, but embedded in code generators and interpreters. Hence, we relax the notion of validity and require the co-evolution specification to be total, *i.e.,* every valid model for the original DSL is mapped to a valid model in the evolved DSL.

Summarizing, we observe that to the best of our knowledge, no implementation for **C6** is available.

Deriving a Co-Evolution Specification: (10,11,13).

Many of the existing solutions for DSL/model co-evolution, derive a co-evolution specification from the corresponding evolution specification [36, 29].

Additionally, Kappel *et al.* [16] have defined a method called "Model Transformation By-Example". The method starts with the user implementing a co-evolution for a couple of models. From these sample co-evolution specifications, they derive a generic co-evolution specification that should be applicable to all models, dropping the need for an evolution specification all together.

However, regardless of the technique chosen, derived co-evolution specification should be total. Otherwise, a co-evolution specification derived might turn out to be useless. Since no implementation of **C6** is available to the best of our knowledge, we consider implementation of component **C7** to be an open question. Additionally, with respect to this derivation, there are other limitations to consider, which we will discuss in Section 5.

[2]Previously known as COPE

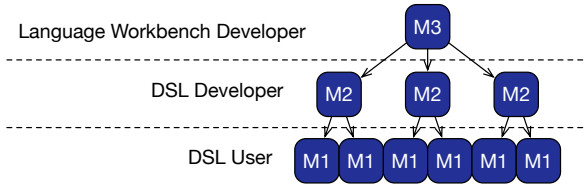


Figure 5: The three levels of meta-modelling, and their available information.

Conclusion.

Having summarized the state-of-the-art for DSL/model co-evolution, we conclude that the following questions should be answered, before the DSL/model co-evolution architecture can be completely implemented:

1. **C2**: How do we specify OCL evolution?
2. **C6**: Is a given co-evolution specification total?
3. **C7**: How do we derive a total co-evolution specification from an evolution specification?

5. THEORETICAL LIMITATIONS TO AUTOMATION

In the previous sections, we have defined a number of components that have to be implemented in order to automate the co-evolution of models with respect to DSL evolution.

Herrmannsdörfer *et al.* have already argued that, in general, no co-evolution specification can fully automate model co-evolution [44]. In the same work, the authors present a solution based on user interaction. However, there are additional limitations not related to user interaction, but related to the information available.

Deriving co-evolution specifications from evolution specifications is hindered by presence of OCL constraints in DSLs. The OCL constraints are not known by the developer of (co-)evolution tooling, as they reside at the level of actual meta-models. We structure the information available to developer of co-evolution tooling by explain the three different levels of information (illustrated in Figure 5).

At **level 1** reside the *Language Workbench Developers*. They have knowledge of a specific meta-meta-model (*e.g.,* `Ecore` [45]), but have no knowledge of specific meta-models (*e.g.,* as the workbench they develop may be used outside their own company). The challenge they face is that any tools, or techniques developed must be generic and applicable to any meta-model conforming to the meta-meta-model.

This means that a researcher wanting to create a complete and reusable piece of evolution tooling must account for every possible meta-model, and every combination of possible OCL constraints on that meta-model. With respect to co-evolutions for these evolutions, every valid instance of that DSL (*i.e.,* any possible meta-model with any valid combination of OCL constraints) should be accounted for. We deem that creating reusable pieces of tooling at this level is, therefore, unfeasible. What remains is to assist the developers at the next level in creating good co-evolution specifications.

At **level 2** reside the *DSL Developers*. DSL developers have knowledge of their own specific meta-model including the OCL constraints present. Additionally the DSL developer has access to the evolution specification of that meta-model and its OCL constraints. However, the DSL developer may still have no knowledge of which instances (models) of their DSL actually exist (*e.g.,* because the models are made at external companies), and must thus (in their work) account for all possible models. However, tooling exists to give a formal definition of what a valid model is (*e.g.,* using EMFtoCSP [40]). In this sense, we could help the DSL developers gain insight into the models that could exist

At the lowest level, **level 3**, reside the DSL users. These users actually create models. At this level there is knowledge of all levels (models, meta-models and meta-meta-models) and the evolution of meta-models. However, creating tooling here is not feasible, as it would have to be re-created for every individual version of every individual DSL.

As, at **level 1** there is not enough information to create static reusable pieces of (co-evolution) knowledge. One can never give a fixed mapping from an evolution specification to a co-evolution specification that works for every meta-model, because for every such mapping there is a possible OCL constraint that can contradict the mapping.

A solution would be to create a function that, given a DSL (meta-model + OCL) and an evolution specification, yields a co-evolution specification. Such a function would have to account for every possible meta-model, and every possible combination of OCL constraints on that meta-model. At present, we do not see how to approach this problem.

We thus believe that creation of the evolution-to-co-evolution mapping should be carried out at **level 2** (DSL developer), as the DSL developers do have knowledge of which OCL constraints should be accounted for. The next obvious goal should thus be to support the DSL developer in creating a valid mapping (*e.g.,* by creating counter examples of models that are not validly co-evolved for a given mapping).

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented a high-level architecture for implementing a tool that automates model co-evolution with respect to DSL evolution to the highest degree possible. Based on this architecture, we have presented a number of questions that have to be answered before the architecture can be fully implemented.

Furthermore, we have looked at the state of the art in DSL/model co-evolution to assess to what extent the posed questions have already been answered. We concluded that specification of both meta-model evolution, and model co-evolution are supported by existing formalisms, but the specification of OCL evolution is not. Furthermore, we observe that there is no formal check whether a given co-evolution specification is valid, and that this question has to be answered before co-evolution specifications can be derived from evolution specifications.

As future work, we consider formal modeling of co-evolution specifications and checking whether their validity. Additionally, we are considering specification of OCL evolution as future work.

7. REFERENCES

[1] M. Herrmannsdörfer, S. Benz, and E. Juergens, "Automatability of coupled evolution of metamodels and models in practice," in *MoDELS*, ser. LNCS. Springer, 2008, vol. 5301, pp. 645–659.

[2] R. R. H. Schiffelers, W. Alberts, and J. P. M. Voeten, "Model-based specification, analysis and synthesis of servo controllers for lithoscanners," in *6th International Workshop on Multi-Paradigm Modeling*. ACM, 2012, pp. 55–60.

[3] M. Herrmannsdörfer, D. Ratiu, and G. Wachsmuth, "Language evolution in practice: The history of GMF," ser. LNCS, vol. 5969. Springer, 2009, pp. 3–22.

[4] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley, 2009.

[5] T. Kühne, "Matters of (meta-) modeling," *Software & Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[6] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Addison-Wesley, 2003.

[7] "OMG SysML," http://www.omgsysml.org/, accessed: 2016-07-05.

[8] M. Lungu, "Towards reverse engineering software ecosystems," in *ICSM*, 2008, pp. 428–431.

[9] A. Serebrenik and T. Mens, "Challenges in software ecosystems research," in *ECSA Workshops*, I. Crnkovic, Ed. ACM, 2015, pp. 40:1–40:6.

[10] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik, "Studying evolving software ecosystems based on ecological models," in *Evolving Software Systems*, T. Mens, A. Serebrenik, and A. Cleve, Eds. Springer, 2014, pp. 297–326.

[11] J. Businge, A. Serebrenik, and M. G. J. van den Brand, "An empirical study of the evolution of eclipse third-party plug-ins," in *IWPSE-EVOL*, A. Capiluppi, A. Cleve, and N. Moha, Eds. ACM, 2010, pp. 63–72.

[12] J.-M. Favre, "Languages evolve too! changing the software time scale," in *Principles of Software Evolution*, 2005, pp. 33–42.

[13] M. Herrmannsdörfer and G. Wachsmuth, "Coupled evolution of software metamodels and models," in *Evolving Software Systems*. Springer, 2014, pp. 33–63.

[14] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*, ser. LNCS. Springer, 2007, vol. 4609, pp. 600–624.

[15] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *IEEE Enterprise Distributed Object Computing Conference*, 2008, pp. 222–231.

[16] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Model

transformation by-example: A survey of the first wave," in *Conceptual Modelling and Its Theoretical Foundations*, ser. LNCS. Springer, 2012, vol. 7260, pp. 197–215.

[17] L. M. Rose, A. Etien, D. Mendez, D. S. Kolovos, F. A. C. Polack, and R. F. Paige, "Comparing Model-Metamodel and Transformation-Metamodel Co-evolution," in *Model and Evolution Workshop*, 2010.

[18] B. Gruschko, D. Kolovos, and R. Paige, "Towards synchronizing models with evolving metamodels," in *Workshop on Model-Driven Software Evolution*, 2007.

[19] A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai, "Automatic domain model migration to manage metamodel evolution," in *MoDELS*, ser. LNCS. Springer, 2009, vol. 5795, pp. 706–711.

[20] J. García, O. Diaz, and M. Azanza, "Model transformation co-evolution: A semi-automatic approach," in *SLE*, ser. LNCS. Springer, 2013, vol. 7745, pp. 144–163.

[21] T. Levendovszky, D. Balasubramanian, A. Narayanan, and G. Karsai, "A novel approach to semi-automated evolution of dsml model transformation," in *SLE*, ser. LNCS. Springer, 2010, vol. 5969, pp. 23–41.

[22] D. Di Ruscio, R. Lämmel, and A. Pierantonio, "Automated co-evolution of GMF editor models," in *SLE*, ser. LNCS. Springer, 2011, vol. 6563, pp. 143–162.

[23] D. Di Ruscio, L. Iovino, and A. Pierantonio, "Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems," in *Graph Transformations*, ser. LNCS. Springer, 2012, vol. 7562, pp. 20–37.

[24] J.G.M. Mengerink, R.R.H. Schiffelers, A. Serebrenik, and M.G.J. van den Brand, "Evolution specification evaluation in industrial mdse ecosystems," Eindhoven University of Technology, Tech. Rep. CSR-15-04, 2015. [Online]. Available: https://pure.tue.nl/ws/files/3757969/390954927658277.pdf

[25] F. M. Haney, "Module connection analysis: A tool for scheduling software debugging activities," in *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, ser. AFIPS '72 (Fall, part I). ACM, 1972, pp. 173–179.

[26] S. S. Yau, J. S. Collofello, and T. MacGregor, "Ripple effect analysis of software maintenance," in *Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International*, 1978, pp. 60–65.

[27] T. Zimmermann, V. Dallmeier, K. Halachev, and A. Zeller, "erose: guiding programmers in eclipse," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005, pp. 186–187.

[28] R. Jongeling, "Change Impact Analysis in Model Driven Software Engineering Ecosystems," Master's thesis, Eindhoven University of Technology, the Netherlands, 2016. [Online]. Available: http://alexandria.tue.nl/extra1/afstversl/wsk-i/Jongeling_2016.pdf

[29] J. Di Rocco, L. Iovino, and A. Pierantonio, "Bridging state-based differencing and co-evolution," in *Workshop on Models and Evolution*. ACM, 2012, pp. 15–20.

[30] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "An analysis of approaches to model migration," in *MoDSE-MCCM Workshop*, 2009, pp. 6–15.

[31] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning, *The State of the Art in Language Workbenches*. Springer, 2013, pp. 197–217.

[32] J. G. M. Mengerink, A. Serebrenik, R. R. H. Schiffelers, and M. G. J. van den Brand, "A complete operator library for DSL evolution specification," in *ICSME*, 2016.

[33] "QVT," http://www.omg.org/spec/QVT/, accessed: 2015-04-07.

[34] "QVTo," http://www.eclipse.org/mmt/?project=qvto, accessed: 2015-04-07.

[35] "EMF Compare," https://www.eclipse.org/emf/compare/, accessed: 2015-04-07.

[36] "Edapt," https://www.eclipse.org/edapt/, accessed: 2015-04-07.

[37] Y. Vissers, J. G. M. Mengerink, R. R. H. Schiffelers, A. Serebrenik, and M. Reniers, "Maintenance of specification models in industry using Edapt," in *FDL*, 2016.

[38] J. G. M. Mengerink, A. Serebrenik, R. R. H. Schiffelers, and M. G. J. van den Brand, "Udapt: Edapt extensions for industrial application," in *IT.SLE*, ser. CEUR-WS, 2016.

[39] E. Burger and A. Toshovski, "Difference-based Conformance Checking for Ecore Metamodels," in *Proceedings of Modellierung 2014*, ser. GI-LNI, vol. 225, 2014.

[40] C. A. González, F. Büttner, R. Clarisó, and J. Cabot, "Emftocsp: A tool for the lightweight verification of emf models," in *Formal Methods in Software Engineering: Rigorous and Agile Approaches*, ser. FormSERA '12. IEEE, 2012, pp. 44–50.

[41] "UML," http://www.uml.org/, accessed: 2016-06-28.

[42] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[43] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. Polack, "Model migration with Epsilon Flock," in *Theory and Practice of Model Transformations*, ser. LNCS. Springer, 2010, vol. 6142, pp. 184–198.

[44] M. Herrmannsdörfer and D. Ratiu, "Limitations of automating model migration in response to metamodel adaptation," in *MSE, Workshops and Symposia at MODELS*, ser. LNCS, vol. 6002. Springer, 2009, pp. 205–219.

[45] "Ecore," lhttp://www.eclipse.org/modeling/emf/, accessed: 2016-7-20.

Software Evolution Management: Industrial Practices

Antonio Cicchetti
Mälardalen University, IDT
72123, Västerås, Sweden
antonio.cicchetti@mdh.se

Federico Ciccozzi
Mälardalen University, IDT
72123, Västerås, Sweden
federico.ciccozzi@mdh.se

Jan Carlson
Mälardalen University, IDT
72123, Västerås, Sweden
jan.carlson@mdh.se

ABSTRACT

The complexity of modern software systems and the global competition make the adoption of model-based techniques unavoidable. A higher level of abstraction not only allows to mitigate the intricacy of the development, e.g., through separation of concerns, but it is also expected to permit shorter round-trip cycles to add new system functionalities, fix bugs, and refine existing features.

This paper reports practical experiences in the management of industrial software evolution collected by means of semi-structured interviews with software development experts. All the interviewed companies develop embedded real-time safety-critical systems and aim at reaching more agile processes. Interestingly, while model-based methodologies appear to be widely accepted, shortening round-trip cycles due to changes appears still to be a major issue towards a more efficient development process.

Keywords

Industrial software systems; Model-driven engineering; Model-based development; System evolution; Round-trip engineering

1. INTRODUCTION

Nowadays software can be practically considered as part of systems in any application domain. It is typically exploited to enhance or even substitute electrical and mechanical portions of systems in order to make the final product more efficient, durable, and appealing from a user's perspective. In other words, software is used to *smartify* systems by adding complex features thanks to its malleability. This wide adoption does not exclude mission-critical systems, that is systems for which a failure can have huge impacts in terms of money loss and harm to human lives. In this respect, the development of modern industrial systems is an endeavour that has to trade off development costs, time-to-market, system reliability, and customers' satisfaction, just to mention a few.

Model-Driven Engineering (MDE) [14] aims at alleviating the complexity of building modern software systems by advocating a model-centric development paradigm. Models permit to abstract away the unnecessary details and to focus on the aspects that matter in a particular development stage and/or in a specific-domain [3]. Moreover, models disclose unprecedented opportunities for automation, as they can be exploited to perform early system analysis and validation, as well as contribute to the creation of the final production code [9]. However, empirical studies have stressed that one of the major obstacles to the adoption of MDE in industry is the lack of adequate tool support [17, 5, 9], which is often intended as the need for custom modelling languages and notations, analysis and validation facilities, code generation transformations, tools able to deal with complex systems, and so forth [13].

This paper describes the preliminary exploration of industrial practices related to the evolution of software-related artefacts, meant as models, documents and production code, throughout the development process. The aim is to identify the most relevant factors hampering flawless evolution and thereby an efficient iterative approach. The general common development traits emerged from this investigation are:

- all interviewed companies use a centralised document repository which keeps track of development versions;
- the development flow is top-down, meaning that very little information on changes is propagated from lower levels of abstraction up to higher levels;
- the development process is horizontally iterative but vertically waterfall, meaning that once artefacts at a certain level of abstraction are considered successfully completed, they are frozen until the whole process is iterated again.

The core contribution of the paper is meant to be a set of findings concerning the main issues with current industrial practices when it comes to evolution of software-related artefacts. In addition, we provide what we believe are interesting research directions towards tackling the identified challenges. Despite the collected results not constituting statistical evidence, the interviews remark the need of a proper tool chain able to guarantee the consistency across development stages and artefacts as the precondition to achieve MDE promised gains. Otherwise, separation of concerns ends up in creating several *discontinuities* [15] that jeopardise the consistency of the integrated system and slows down the development process. In fact, not only closing those discontinuities is a time-consuming and error-prone task, but the existing gaps hamper any opportunity of automating change impact analysis, thus limiting the chances of reasoning about system evolution and its side-effects.

The paper is organised as follows: the next section provides background details about the context in which the investigation has been performed, while Section 3 discusses how the interviews have been structured and done. Section 4 provides a deep analysis about the findings that we could extract from the data gathered via interviews and Section 5

describes our view on how to proceed further for mitigating identified challenges. Eventually, Section 7 discusses related investigations on the application of MDE in industrial contexts while Section 8 draws conclusive remarks about the current work.

2. CONTEXT

The development of software shows a trend of increasing complexity under ever tighter time and budget constraints. In order to face this scenario, the development process has necessarily to be as swift as possible and avoid possible stalls due to inefficiency. Only in this way, product development can promptly adapt to the continuously changing market demands. In this respect, agile techniques promise to improve development by reducing the time spent on requirements gathering and analysis, and in general by reducing documentation efforts to a minimum [10].

Software Center (SwC)[1] is a joint effort of Swedish industry and academia which aims at rapidly introducing innovation in industry with the support of academic partners.

The results illustrated in this work have been collected in the context of a SwC project[2], which addresses the support of architectural artefacts evolution; the authors of this paper represent the research team in the project. The aim is to shorten system development iterations by means of an adequate change impact analysis and tracing of the propagation of system evolution effects. The first sprint was intended to investigate the current state-of-practice inside the participating companies, in order to better understand their development processes, evolution pressures, solutions, and needs. This paper describes the outcome of this first sprint in terms of industrial practices related of evolution of software-related artefacts.

3. SEMI-STRUCTURED INTERVIEWS

The project started with a preliminary workshop in which the participating companies and the researchers had an open discussion about the topic in order to align the different points of view and interest. In the workshop, the companies gave a high-level description of their current situation in terms of architecture/design activities and the tool chains supporting them with particular focus on how evolution is handled and on migration of information between different tools, formalisms and storages. The outcome of the workshop was that evolution management and migration of various software-related artefacts was the common point of interest. To gather more specific data on the practices and problems at each company, we decided to carry out a set of semi-structured interviews with companies staff. This data was analysed and represents the ground for the next project iterations. In this section we describe how we set up and carried out the semi-structured interviews.

3.1 Interviews preparation

Data was gathered through semi-structured interviews [8] with staff. The difference between structured and semi-structured interviews is that in the latter the interviewee is allowed to divert from an initial set of open questions, which are considered as a minimal set of well-thought topics, related to software evolution in our case, to be discussed. Such topics (and questions) were used by the interviewers as interview guide, and were sent in advance to the interviewees for preparation. During the interviews, questions were asked in different ways and at different moments, depending on the interviewee and the flow of discussion. Doing so, we were able to customise our questions to the specific interview and interviewee.

The set of topics and questions was the following:

- T1 Tools ecosystem – What formalisms, languages and tools do you use for development and documentation of design/architecture/implementation?
- T2 Artefacts storage – How are artefacts (models/code/documents) stored?
- T3 Evolution strategies – How, and how often, do you exchange/migrate/synchronise/version information between different languages/tools/formats/storages?
- T4 Diff/merge and conflict resolution – When exchanging/migrating/synchronising/versioning artefacts, how is differencing and merging as well as resolution of conflicts among artefacts handled?
- T5 Concurrent development – How common is it that multiple persons work on the same artefact (concurrently or interleaved) and how do you currently organise concurrent work?
- T6 Envisioned improvements – What improvements would you like to see in relation to your current evolution management practices?

Semi-structured interview was also chosen as data collection method since, being at an initial phase of investigation, we did not want to steer the interviews in a predefined direction through a strict questionnaire. On the contrary, with semi-structured interviews, we wanted to gather a broader set of information about practices, problems, ideas about software evolution in the companies.

3.2 Running the interviews

We interviewed a total of 9 individuals across the 3 companies (3 persons per company); interviewees were chosen so to represent a reasonable range of different roles involved in software development. The three companies are large Swedish enterprises with focus on embedded real-time and safety-critical systems in different application domains. We interviewed software engineers, testers, product managers, and modellers. All interviewees had experience with MDE. Interviews were conducted on-site by at least two of the authors and in the interviewees' native language, Swedish. The set of questions was sent to the interviewees in advance in order to allow them to raise possible issues and prepare for the interview by gathering additional information if needed.

The interviews started with a short summary by the interviewers about the SwC project and the interview itself. This was followed by the interviewee describing shortly her current role in the company and experience with similar topics in general. Thanks to the fact that we exploited a semi-structured format, we could notice that the various interviews took different, and interesting, routes, highlighting expected and unexpected problems in the industrial software

[1] http://www.software-center.se/.
[2] http://www.software-center.se/research-themes/technology-themes/continuous-architecture/Evolution+Support+for+Architectural+Artefacts.

evolution. In case of unclear statements by the interviewee, a clarification was always sought and given before going forward to the next topic.

The interviews lasted between 30 and 90 minutes and were recorded in order for resulting data to be analysed and synthesised. For analysis purposes, the recorded audio of each interview was listened to by one researcher and transcribed by question; additional details not related to a specific question were annotated separately. The transcribed answers were discussed extensively within the research team in order to identify the most relevant findings to document and that would represent the basis for the next project iterations.

4. RESULTS

The interviews gave interesting answers to the posed questions but also provided unexpected insights thanks to the semi-structured format. In this section we provide a summary of the most relevant (shared) findings that the research team identified by discussing the transcribed interviews.

4.1 Tools ecosystem and artefacts storage

The answers gathered during the interviews clearly show that at companies there usually exists a plethora of tools, languages and formalisms used more or less together for requirements specification, software modelling and production code. Even if standard languages (e.g. ADLs or UML) are used, the observed trend is working on company-tailored versions of existing commercial tools. Moreover, different groups in the same company use different toolchains, even in joint projects. When it comes to requirements, the common line is represented by textual descriptions that are maintained in various tools. These textual descriptions are manually broken down to a so called "software architecture" defined in terms of graphical models. The architecture usually defines parts of the software and hardware nodes composing the entire system as well as allocation of software to hardware, and it is used for early assessment of modelled functionalities. In some cases, dynamic parts of the software system are modelled using tools and languages different from the ones used for architectures. Moreover, in many cases different parts of the software system are modelled and developed with different tools and languages. When it comes to production code, in some cases it is generated from models, while in others models are just used as blueprint for the programmers to implement. The common trait is that consistency models-code is regarded as important but currently hard to achieve since it is not automated (many tools in the ecosystem do not have automatic bridges): *"It happens that, after a number of years, models are not updated anymore, thus losing their consistency with up-to-date code.. There is simply no time to do it.. It feels like the lack of agile ways to evolve models and related artefacts leads to neglect models and update only the code.*[3] *".*

Models, documentation and production code are managed with different tools but are commonly stored in a central database. Anyhow, having them in a central database does not provide tangible advantages when it comes to evolution since there are still serious issues in bridging the various tools: *"Exchanging information among tools using XMI creates many problems due to the format misalignment in the various tools..".*

In synthesis, many heterogeneous tools and languages are used without effective and automated mechanisms for univocally going from one to the other and for ensuring that changes to one artefact are correctly propagated to related and dependent artefacts. This makes evolution of the various artefacts (documents, models, code) a manual, error-prone, tedious and in some cases a shabby task.

4.2 Evolution strategies, diff/merge and conflict resolution

Evolution strategies, especially among related artefacts of different types, are shallow. Changes to code are not always reproduced in models and documentation since it is a manual effort up to a single individual and there does not seem to be a strong focus on "consistency maintenance" of less crucial artefacts (e.g., models and documentations) in the companies. Running versions of code on products with very long lifetime are kept as they are as much as possible; only small fixes are made when unavoidable. Moreover, evolution happens at different levels and in different tools although affecting the same artefact; while some automation for propagating change from one level and/or one tool to the other exists, much is still left to manual, unstructured activities. Instead of improving evolution strategies, all companies try to minimise the need of evolution by going for a conservative and add-only strategy as much as possible.

There is usually no efficient support for round-tripping from code to models. More specifically, companies would find it beneficial to propagate changes done on code back to models and documentation (across different tools), for instance in terms of warnings. The impossibility to efficiently do this, as well as the aforementioned intrinsic difficulties in bridging the many tools composing the ecosystem, *".. make often software development start from scratch in new projects rather than reusing models, documents and code (or parts of them) from previous successful projects."*. Instead, new projects should be intended as evolutions of previous ones, with artefacts that evolve from one project to the other in a unbroken manner.

The common trait with diff/merge is that all companies try to avoid merging as much as they can. Differencing is, on the other hand, considered a very important aspect but, being often a manual task, it is error-prone and tedious. Implicit diff/merge is done by versioning tools used by the companies, such as Subversion; also in this case, several issues arise when versioning graphical models exploiting their textual representation. Moreover, the heterogeneity of formats among the many different tools and shaky links among them clearly does not simplify diff/merge and conflict resolution.

4.3 Concurrent development

Concurrent modelling is commonly achieved by partitioning models in sub-portions (either physical files or logical sub-systems) in order to avoid concurrent changes on the same model portion. While developers do not work with the same model portion, they usually work on the same database concurrently and the data is continuously synchronised through automatic database-specific mechanisms so that everyone has the same overall picture of the up-to-date product version. There does not seem to be any particular problems with this practice; this is explained by the fact

[3]Note that reported extracts from interviews have been translated by the authors from Swedish to English.

that in reality no concurrent modelling ever takes place since sub-portions are not meant to be concurrently accessed by multiple persons. Nevertheless, the possibility of disruptive changes to dependent software modules is still possible both at model and code level due to (i) human misunderstandings and (ii) because no dependency analysis is done among different software modules. *"These kind of problems do not come up until testing production code.. for instance by getting compilation errors."*.

4.4 Envisioned improvements

An aspect which seems to be of high priority for the interviewed companies is the need for improving traceability, that is to say back-tracing of changes done at lower levels of abstraction (e.g., code) to related artefacts at higher abstraction levels (e.g., models and documents). This is seen as crucial for being able to enhance traceability as well as improving versioning and boosting reuse. Particularly important is considered to achieve much more efficient (and automated) ways to ensure consistency between models and code in the long run. From our interviews it became clear that practitioners are starting to perceive the current fixed tool-centric idea as somewhat obsolete within a model-driven or model-based development process. Instead, they put emphasis on the need to move the attention towards more flexible (meta)model-centric approaches. Functionalities get more in number and more complex; additionally, there is a much higher dependability among different parts of the software system than before. This complicates evolution, also considering the huge variations among product versions; legacy running systems must be supported and updated in a smart, non-breaking way. Having a (meta)model-centric approach would permit to *".. define variation points in the metamodel that would simplify evolution tasks."*.

5. DISCUSSION

So far we described the feedbacks collected during the interviews with companies, grouped by topic. Based on those feedback, this section highlights a set of relevant issues that we believe hamper a better management of evolution, together with a set of possible corresponding research investigation directions in order to improve the current practice.

A critical aspect that seems to be underestimated by current empirical investigations on the adoption of MDE in industry is the transitional nature of such a process [2]. In fact, usually software is developed by companies whose main products are not software (e.g., cars, satellites). Therefore, software development gets intertwined with other engineering activities and very often imposing a one-step adoption of a completely (even if fully featured) new development platform is not realistic.

Challenge 1: Heterogeneity of tools and languages in the toolchain

Issue Heterogeneity of the many (modelling) tools and languages typically exploited in an industrial development process is a major hinder for effective evolution due to the lack of appropriate format exchange and change propagation support.

Investigation directions Introduce more powerful and automated links between interconnected artefacts across different tools and languages, and in the long run create bidirectional bridges (through e.g. model transformations) supporting uncertainty.

A more realistic scenario is a step-wise adoption of domain- or task-specific tools that contribute to the development process. The process is orchestrated by means of a centralised storage support, which is format-agnostic, and where tool interconnections are kept through links. Since in general the tools cannot communicate with each other, change propagation can only be supported in its minimal terms, that is raising warnings for artefacts linked to entities involved in evolution activities. Moreover, consistency management becomes necessarily a manual task.

The companies solve consistency and change propagation issues by constraining the development in a waterfall process. More precisely, all the artefacts related to a certain development step or abstraction level (that is, horizontally) are iteratively developed until they satisfy a planned goal. After that, those artefacts are *frozen*, i.e. kept as read-only until the next development process iteration. In this way, consistency between abstraction levels is enforced by the subsequent steps, since each higher abstraction level is input for lower ones. Besides, whenever there is a need to make changes outside the specification coming from a higher level of abstraction, this would be recorded in terms of a change request to be dealt with in the next iteration.

Given this scenario, we propose two possible investigation directions: introducing more powerful links in the central storage and/or creating transformations between tools. The former can be conceived as a short term solution, since it does not require many changes in an existing development process. However, already by adding semantic information to the links (notably, the degree of dependence between linked artefacts) it would improve change impact analysis. This solution would require a company-specific investigation of the evolution characteristics that artefacts undergo during a typical development process.

In the long run, tools should be connected by means of model transformations. These transformations should be bidirectional to allow the synchronisation on both sides, and supporting uncertainty to be able to effectively deal with the intrinsic heterogeneity of the mappings [6, 7]. In this case, the degree of interconnection between the various tools should be carefully evaluated, in order to avoid the need of writing dozens of model transformations.

Challenge 2: Diff/merge at the appropriate abstraction level

Issue Differencing and merging are mostly manually performed (when they cannot be avoided) since currently employed versioning tools are text-based and not able to effectively operate diff/merge on complex artefacts, such as graphical models. The XMI format does not seem to help in this matter.

Investigation directions Introduce diff/merge operations at the modelling level of abstraction.

Differencing and merging models at an appropriate level of abstraction is a known problem for the MDE research community. The proliferation of tools illustrated in the previous challenge stresses this need since manipulations might have different interpretations/impacts depending on the consumer of the modified artefact. Moreover, in general modelling tools are not equipped with diff/merge features.

Companies tackle this issue by avoiding diff/merge, i.e. by *serialising* concurrent manipulations through the exploitation of disjoint sub-portions of models and/or the adoption of artefact locks. Even more, in some cases code is edited by hand instead of re-generated after an appropriate modification of the source models it originates from, especially in case of small refinements. This relieves companies from making an additional effort in understanding how model changes would propagate to interconnected artefacts at higher abstraction levels.

For this challenge we propose the study of appropriate diff/merge mechanisms. The solutions could be gradually developed and introduced firstly based on the improved links discussed in the previous challenge, and hence trying to detect at least the changes relevant for tool interconnections. In the long run, change detection should provide enough information to enable a reliable change impact analysis and possibly automated propagation through synchronisation transformations.

Challenge 3: Explicit traceability between related artefacts

Issue The lack of effective support for explicitly keeping track [1] of dependencies among different artefacts in the many tools makes round-trip engineering very difficult and error-prone.

Investigation directions Analyse the changes at various levels of abstraction, at different development stages, with respect to the used tools, etc., and classify them in terms of their impact on other artefacts. In the long run, produce estimates of propagation effort due to changes.

The central storage used to coordinate the development process can be considered as a very basic support for traceability. However, the lack of adequate interconnection information makes this tracking support not effective. Notably, the dependencies among different artefacts in the many tools can make cautious concurrent modelling still cause disruptive changes that can go unnoticed until production code is tested. In turn, this slows down the whole process due to the efforts required to understand the origins of the problem.

Another important issue caused by the lack of traceability is the poor support for round-trip engineering. Since the link between the various artefacts gets typically blurry when going from an abstraction level to another, it is very difficult to understand how modifications at lower abstraction levels should be propagated back to higher levels. The companies mitigate this issue by adopting the waterfall-like process described before, which however slows down the development process and limits the opportunities for concurrent development.

Our proposal is to empirically retrieve historical evolution information and propose a characterisation of possible system evolutions. The improved interconnections between the tools proposed for *challenge 1* would support a more effective synchronisation of artefacts at different levels of abstraction. Together, these two investigations can create the potentials for an enhanced round-trip engineering process, in which code and other artefacts can be consciously modified, and the effects of the modifications can be better analysed and propagated to interconnected artefacts.

6. THREATS TO VALIDITY

When it comes to internal validity, we adopted a systematic approach in preparing the study, gathering, analysing and synthesising data. On the one hand, since we did not opt for structured interviews with a strict set of questions, but rather went for semi-structured interviews, the rigour of the study and its results could be questioned. On the other hand, less structured methods [8] have already proven very useful when the interviewer seeks a broad spectrum of information. This is possible since the interviewee is given more freedom and somehow participates actively to steer the discourse even towards unforeseen (but not less interesting) directions [12]. In order to avoid misinterpretations of answers given by interviewees, interviews were always carried out by at least two members of the research team. Moreover, analysis and synthesis of the gathered data as well as elicitation of the findings were performed by the entire research team.

Regarding external validity, the interviewed companies develop large-scale software to run on safety-critical embedded real-time systems with long lifetime. This means that our results can happen to not being applicable to very small business cases or short lifetime products. Moreover, due to the fact that we did not adopt a strictly structured interview method and given the rather limited number of interviewees we do not claim statistical relevance of our results. Nevertheless, we believe that the results coming out from our study, especially considering the fact that they were agreed by all interviewed companies, represent a first step towards understanding the practical issues of evolving software artefacts when adopting model-driven or model-based development.

7. RELATED WORKS

The last decade has seen an increasing amount of publications devoted to empirically assess the industrial practices in adopting MDE and both its good and bad effects [11, 9]. Usually the existing literature surveys the problem from an overall software development process perspective, analysing the effects across different applicative domains [5, 17, 4] and/or target systems [9]. On the contrary, this work specifically targets evolutionary scenarios and the impact of MDE in their management. In this respect, as mentioned in Section 5, even if our data cannot be considered as statistically relevant, it is possible to notice some trends confirming other existing results, and also to deduce interesting explanations about some current challenges in adopting MDE in concrete

industrial settings.

Burden et al. [5] present a study of MDE adoption at three large companies, two of which are also involved in this work. It is therefore not surprising that we share several observations with the cited paper, especially related to the development process and the need of freezing artefacts/specifications pertaining to an abstraction level (or development stage) before proceeding further. More generally, studies like [5, 4, 16] or the one described in this work remark the distinction between companies producing software as main business and companies using software in their products: in particular, the latter typically face more problems due to the need for integration of the software development with the remaining part of the system realisation process.

It is worth noting that most of the empirical investigations seem to assume the adoption of MDE as a one step process, or they observe the effects once the adoption process is considered as satisfying/completed. However, MDE adoption usually happens as a transition [2] in which MDE methods are incrementally *plugged in* in the development process. This also emerges in our interviews, where there exist several degrees of adoption even when considering different departments in the same company. The consequence is that issues due to the partial adoption of MDE are revealed through other side-effects and troubles, notably the lack of modelling competence, the inadequacy of the tools and their integration, the extra efforts required to boot-strap MDE-based development processes.

8. CONCLUSIONS

This paper discussed the issues faced in the management of evolution in industrial software development processes adopting MDE. These issues have been identified by analysing the data gathered from semi-structured interviews with practitioners in the context of an industry-academia joint research project aiming at enhancing development processes.

The problems described by the interviewees remarkably affect the development performances and hamper effective round-trip engineering. Starting from those problems, in this paper we list a number of relevant challenges together with their effects on the development process. Moreover, we propose a set of corresponding research directions that we plan to investigate as next steps in the joint research project mentioned above.

9. ACKNOWLEDGMENTS

The authors would like to thank all the interviewees that accepted to collaborate in our project and provided useful and honest insights about the internal development practices at their companies. The project is supported by Software Center.

10. REFERENCES

[1] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[2] J. Aranda, D. Damian, and A. Borici. *Transition to Model-Driven Engineering*, pages 692–708. Springer, Berlin, Heidelberg, 2012.

[3] J. Bezivin. On the Unification Power of Models. *SoSym*, 4(2):171–188, 2005.

[4] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz. *What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?*, pages 343–369. IGI Global, 2012.

[5] H. Burden, R. Heldal, and J. Whittle. Comparing and contrasting model-driven engineering at three large companies. In *Procs. of the 8th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 14:1–14:10, New York, NY, USA, 2014. ACM.

[6] R. Eramo, A. Pierantonio, and G. Rosa. Managing uncertainty in bidirectional model transformations. In *Procs. of the 2015 ACM SIGPLAN Int. Conf. on Software Language Engineering*, SLE 2015, pages 49–58, New York, NY, USA, 2015. ACM.

[7] C. Hardebolle and F. Boulanger. Exploring multi-paradigm modeling techniques. *SIMULATION*, 85(11-12):688–708, 2009.

[8] S. E. Hove and B. Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 10–23. IEEE, 2005.

[9] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, pages 1–23, 2016.

[10] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[11] P. Mohagheghi and V. Dehlen. *Where Is the Proof? - A Review of Experiences from Applying MDE in Industry*, pages 432–443. Springer, Berlin, Heidelberg, 2008.

[12] K. Musante and B. R. DeWalt. *Participant observation: A guide for fieldworkers*. Rowman Altamira, 2010.

[13] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. C. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, J. Kienzle, M. Schöttle, F. Steimann, D. Stikkolorum, and J. Whittle. *The Relevance of Model-Driven Engineering Thirty Years from Now*, pages 183–200. Springer International Publishing, Cham, 2014.

[14] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[15] B. Selic. The Pragmatics of Model-driven Development. *IEEE Software*, (5):19–25, 2003.

[16] M. Staron. *Adopting Model Driven Software Development in Industry – A Case Study at Two Companies*, pages 57–72. Springer, Berlin, Heidelberg, 2006.

[17] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. *Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?*, pages 1–17. Springer, Berlin, Heidelberg, 2013.

Automatic Change Recommendation of Models and Meta Models Based on Change Histories

Stefan Kögel, Raffaela Groner, and Matthias Tichy
Institute of Software Engineering and Programming Languages
Ulm University
D-89069 Ulm
[stefan.koegel|raffaela.groner|matthias.tichy]@uni-ulm.de

ABSTRACT

Model-driven software engineering uses models and meta models as key artefacts in the software development process. Typically, changes in the models (or meta models) do not come in isolation but are part of more complex change sets where a single change depends on other changes, e.g., a component is added to an architectural model and thereafter ports and connectors connect this component to other components. Furthermore, these sets of related and depending changes are often recurring, e.g., always when a component is added to an architecture, it is highly likely that ports are added to that component, too. This is similar for changes in meta models. Our goal is to help engineers by (1) automatically identifying clusters of related changes on model histories and (2) recommending corresponding changes after the engineer performs a single change. In this position paper, we present an initial technique to achieve our goal. We evaluate our technique with models from the Eclipse GMF project and present our recommendations as well as the recommendation quality. Our evaluation found an average precision between 0.43 and 0.82 for our recommendations.

Keywords

Model-driven development; Change recommendation; Revision history mining

1. INTRODUCTION

As models are key artefacts in model-driven software engineering, software engineers typically spend much time creating and evolving models. Hence, they need good tool support to efficiently work with models.

There exist many simple or more complex tools in integrated development environments or code editors to improve the productivity of software engineers, for example, auto completion, quick fixes, refactorings, and templates for often used language constructs. These tools aim at improving development speed and quality. However, such tools are typically not available for changing models in model-driven software engineering (with the exception of autocompletion, e.g., in Xtext [6] based textual editors).

In our personal experience in modeling, we often had to perform repetitive and recurring changes when evolving the models. Furthermore, we sometimes forgot some individual changes in a model when performing complex changes.

Our hypothesis is that we can improve modeling speed and quality of model changes by recommending model changes to the engineer based on current changes and historical changes. For example, after adding a transition to a state machine, guards or actions are added to the transition afterwards.

Please note that the same argumentation holds for meta models as well. For example, often meta classes need to subclass a certain superclass. Therefore, after creating the meta class, it might be beneficial to recommend the addition of a generalization relationship to that meta class.

Recommender Systems aim at supporting users in making decisions. They recommend items of interest to users based on explicitly or implicitly expressed preferences [15]. An example of a recommender system in software engineering aims at proposing reuse possibilities in writing test cases [9]. Related to model-driven engineering, Brosch et al. presented a recommender addressing the conflict resolution in merging models [3]. However, there does not exist a recommender system to recommend modeling changes to the engineer as discussed before.

In this position paper, we propose a preliminary approach for generating live recommendations to engineers modifying models. Specifically, the approach recommends model changes to the engineer based on his currently performed changes where the to-be-recommended changes have been linked to the currently performed changes in historical change sets of the model.

The aims of our approach are that it (G1) automatically produces recommendations, (G2) aggregates the recommendations in order to not overwhelm the user with too much information, and (G3) does not make too many wrong recommendations to prevent users from losing confidence.

As the approach requires a set of historical model changes, we mine model changes from version control systems and use SiLift [10, 11] to compute the individual model changes for each version. Our evaluation on several meta model histories show that we can reach medium to high precision.

We describe our approach for automatic change recommendation and several extensions in Section 2. In Section 3, we present an evaluation of our technique using several meta models from Eclipse Projects. Related work is discussed in Section 4. Section 5 concludes our paper and discusses future work.

2. TECHNIQUE

Our goal is to recommend further changes to a model based on current and historic changes. To achieve this goal we use the SiLift Tool [10] to compute the differences between historic versions of the same model. These differences are consistency preserving [11]. Figure 1 gives a high level overview of our technique. Rectangles represent data

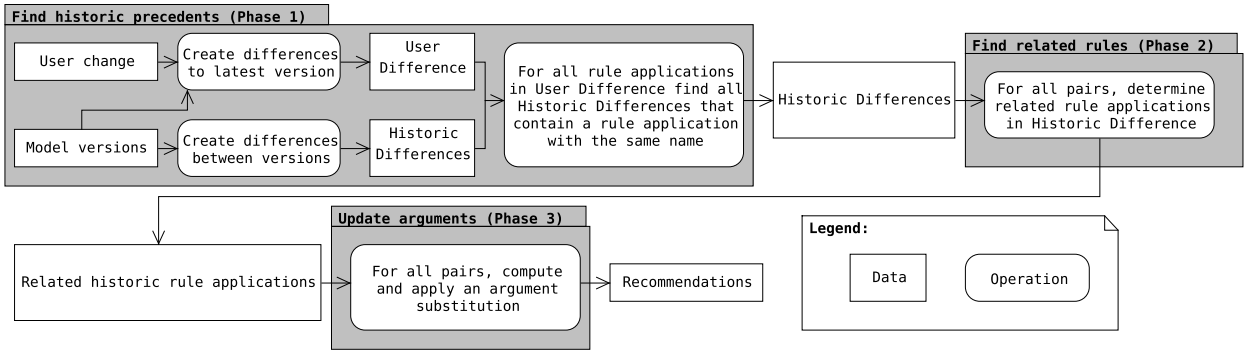


Figure 1: Diagram of our technique

CreateEClassInEPackage	
↳ selectedEObject:	*gmfgraph*
↳ New:	*LayoutRef*

(a) Create a new EClass

AddEClassTgtEClass	
↳ selectedEObject:	*LayoutRef*
↳ NewTarget:	*Layout*

(b) Creates a super class relationship

CreateEAnnotationInEModelElement	
↳ selectedEObject:	*LayoutRef*
↳ New:	*LayoutRef/GenModel*

(c) Create an EAnnotation in an element

CreateEStringToStringMapEntryInEAnnotation	
↳ selectedEObject:	*LayoutRef/GenModel*
↳ New:	*LayoutRef/GenModel/details*

(d) Create string entry in EAnnotation

Figure 2: Excerpt from a difference with four Henshin rule applications and some of their arguments.

and rounded blocks represent operations on this data. We have divided our technique into three phases: (1) **finding historic precedents** for current user changes, (2) **finding related rules** for the historic precedents, and (3) **updating the arguments** of the historic precedents so that they match the current user changes. In the following, we will motivate and explain our technique using a running example. Note that the presented technique is a proof of concept and needs to be developed further.

2.1 Running Example

Figure 2 shows an excerpt from the difference between two historic model versions. The figure shows four Henshin [1] rule applictions, that consist of the rule names and a set of named arguments that contain, among other things, references to elements in models. In this example, the user has created an EClass identified by *LayoutRef* in the package *gmfgraph* (Figure 2a), set *Layout* as the superclass of *LayoutRef* (Figure 2b), created an EAnnotation in *LayoutRef* (Figure 2c), and created an entry in the EAnnotation (Figure 2d). Given this historic difference, suppose that a user adds a new EClass to the *gmfgraph* package. Our technique should then recommend that the new EClass should have a super class, an EAnnotation and an entry in this EAnnotation.

2.2 Basic Technique

In the following, M is a model and $M_i, 1 \leq i \leq n$ are its n different versions. The differences produced by SiLift $\delta_{i,i+1}$ contain partially ordered sets of Henshin [1] rule applications (rules($\delta_{i,i+1}$)).

We will now explain our basic technique for making recommendations based on historic differences between model versions.

Phase 1: Find historic precedents: When a user makes a change to the latest model version, we compute the difference $\delta_{current}$ between the latest model version and the current version. Then we look at every Henshin rule application in this difference and try to make a recommendation for it. Let $h_{current} \in \text{rules}(\delta_{current})$ be a current Henshin rule application, for example, the addition of a new EClass to the *gmfgraph* package.

First, we need to find historic rule applications that are related to our current rule application. Thus, we search for all historical differences $\Delta_{h_{current}}$ that contain a rule application with the same name as $h_{current}$:

$$\Delta_{h_{current}} = \{\delta_{i,i+1} | \\ h \in \text{rules}(\delta_{i,i+1}), \text{name}(h) = \text{name}(h_{current}), 1 \leq i \leq n\}$$

The technique will generate at least one recommendation for every historical difference that contains a rule application with the same name as $h_{current}$.

Phase 2: Find related rules: Given a rule application $h_{hist} \in \delta, \delta \in \Delta_{h_{current}}$ with the same name as $h_{current}$, we now have to determine which other rule applications in the same difference are related to it. To do this, we search for all rule applications that have an argument in common with h_{hist}, i.e., they both have a reference to the same model element. Note that the steps in **Phase 2** are repeated for every historical rule application with the same name as $h_{current}$.

$$H_{h_{hist}} = \{h | h \in \delta, \delta \in \Delta_{h_{current}}, \text{args}(h) \cap \text{args}(h_{hist}) \neq \emptyset\}$$

In our running example, we determine that the rule applications in Figures 2a, 2b, and 2c are related because they have *LayoutRef* in common, which is not the case for the rule application in Figure 2d.

Phase 3: Update arguments: Given all the related rule applications $H_{h_{hist}}$ from a historic difference δ, we need to update their arguments so as to fit to the current user

CreateEClassInEPackage	
↳ selectedEObject:	**gmfgraph**
↳ New:	**RealFigure**

(a) Create a new EClass

AddEClassTgtEClass	
↳ selectedEObject:	**RealFigure**
↳ NewTarget:	*Layout*

(b) Creates a super class relationship

CreateEAnnotationInEModelElement	
↳ selectedEObject:	**RealFigure**
↳ New:	*LayoutRef/GenModel*

(c) Create an EAnnotation in an element

Figure 3: Figure 3a is a rule application describing a user change. Figures 3b and 3c are recommendations based on the user change and the history in Figure 2.

change $h_{current}$. We use the current rule application $h_{current}$ and its historic counterpart $h_{hist} \in \delta$ with the same name as a guide to compute a substitution *subst* between their arguments.

$$subst(\text{value}(p_{hist})) = \text{value}(p_{curr}) \text{ where}$$
$$p_{hist} \in \text{args}(h_{hist}), p_{curr} \in \text{args}(h_{current}),$$
$$\text{name}(p_{hist}) = \text{name}(p_{curr})$$

Applying this substitution to all rule applications in $H_{h_{hist}}$ results in a set of updated Henshin rule applications that can be applied to the current model (assuming there are no constraint violations).

$$Recommend(h_{current}, h_{hist}, \delta) = \{subst(h) | h \in H_{h_{hist}}\}$$

This new set of Henshin rule applications is one of our recommendations. Note that our technique makes a recommendation for every historic rule application h_{hist} with the same name as the current rule application $h_{current}$.

In our running example, Figure 3a shows the rule application *CreateEClassInEPackage* with an argument value of **RealFigure**. When we use the rule application as a user change and Figure 2 as historic rule applications, our technique would find the related historic rule applications in Figures 2a, 2b, and 2c, because they have *LayoutRef* in common. Note that the rule in Figure 2d has no arguments in common with the one in Figure 2a and, hence, would not be found. The corresponding substitution would be:

$$subst = [gmfgraph \rightarrow \textbf{gmfgraph}, LayoutRef \rightarrow \textbf{RealFigure}]$$

This would lead to the recommendations shown in Figures 3b and 3c. Note that both recommendations now reference **RealFigure** instead of *LayoutRef*, while they have kept their references to *Layout* and *LayoutRef/GenModel*.

There are two problems with these recommendations: (1) There is no recommendation for Figure 2d, even though it is related to the recommendation in Figure 3c via its *LayoutRef/GenModel* argument. (2) The historic references *Layout* and *LayoutRef/GenModel* are carried over to the recommendation without changes. It is possible that these historic references lead to correct recommendations, for example, if most newly created EClasses are sub classes of *Layout*. But it would still be better if we could control their inclusion via a tuning parameter, e.g., by keeping or replacing them with a placeholder value. In the next section, we will present extensions to our technique that deal with these problems.

2.3 Extensions to our Technique

We have implemented extensions to our technique to improve the quality of our recommendations with respect to our quality goals (G1-G3).

2.3.1 Intersection of recommendations

So far, we have only described how recommendations are generated. If there are multiple applicable recommendations (from multiple historical rule applications of the same rule), we have to aggregate them before presenting them, because showing too many recommendations will frustrate users. A simple solution would be to count recommendations that contain the same rules and to rank them accordingly. This is similar to frequency based completion in [4].

In this extension, we propose a different solution. Because recommendations are sets of rule applications, we can compute the intersection between all recommendations. To do this, we define that two rule applications are equal if their names are equal [1]. In this way, we merge several recommendations, originating from the same current user change, into a single one (G2). Furthermore, by reducing the amount of rule applications in a recommendation, we reduce the chance that a recommendation will recommend changes that are not intended by the user (G3). This extension takes place after **Phase 3**.

As an example, suppose we have three recommendations consisting of rule applications (ignoring arguments):
$R1 = \{addClass, addEdge, deleteClass\}$,
$R2 = \{addClass, addAnnotation, addEdge\}$,
and $R3 = \{addClass, deleteEdge, addEdge\}$
then the intersection of recommendations $R1 \cap R2 \cap R3$ would be $\{addClass, addEdge\}$. Because *addClass* and *addEdge* have always occurred simultaneously in all recommendations, we suspect that they should occur together again. *deleteClass*, *deleteEdge*, and *addAnnotation* would not be presented to the user, because we can not decide which of these rule applications is the most likely to be intended and they have never occurred simultaneously in the recommendations.

A drawback of this extension is that correct recommendations may be discarded. Another problem is that the intersection of all recommendations may be empty, but this is easily detected and we can fall back on a different method of reducing the amount of recommendations.

2.3.2 Free Variables

In our running example, we cannot predict to which element the argument *LayoutRef/GenModel* in Figure 2c/3c should refer. So we extend our substitution *subst* from **Phase 3** to substitute the historic element with a place holder, a free variable. Replacing all references to unpredictable elements in the arguments leads to the recommendation in Figure 4. Note that the rule applications in Figures 4b and 4c now refer to free variables `Free1` and `Free3` instead of keeping their references from Figure 2. Also note, that rule applications in Figures 4c and 4d are now related

[1] We have not yet extended this equality to consider arguments.

by the free variable `Free2`, instead of by *LayoutRef/GenModel* as in Figures 2c and 2d.

The introduction of free variables removes all references that were only part of the historic rule applications, while preserving the relations between the rule applications. This makes recommendations more abstract (G3), but also requires users to manually fill in values for the free variables.

2.3.3 Indirect Relations

In our running example (Figures 2 and 3), we have, so far, only recommended two rule applications (Figures 3b and 3c). Figure 2d is not part of the recommendation, because it does not share an argument with the rule application in Figure 2a.

In order to find more related historical rule applications for recommendations, we also look at indirectly related rules. Two rules are related if they have a model element in their arguments in common. Two rules are indirectly related, if they have no elements in common, but there is a third rule that has arguments in common with both. For example Figure 2a and Figure 2c have the element *LayoutRef* in common, while Figure 2c and Figure 2d have the element *LayoutRef/GenModel* in common.

We implement this in our technique by iterating the search for related rule applications in **Phase 2**. After finding all rule applications that are directly related to h_{hist}, we repeat the search for all rule applications that have been found.

$$H_{h_{hist},0} = \{h | h \in \delta, \delta \in \Delta_{h_{current}}, \text{args}(h) \cap \text{args}(h_{hist}) \neq \emptyset\}$$

$$H_{h_{hist},p} = \{h | h \in \delta, \delta \in \Delta_{h_{current}}, h_{ind} \in H_{h_{hist},p-1}, \text{args}(h) \cap \text{args}(h_{ind}) \neq \emptyset\}$$

This search can be iterated for a user defined amount of time or until a fixed point is reached and no more indirectly related rule applications can be found.

Indirect relations introduce many rule applications whose arguments will not be changed by the argument substitution *subst*, because the indirectly related rules have no arguments in common with h_{hist}. This will lead to the recommendation of rule applications with many historical arguments. This can be prevented by using free variables.

This extension can also lead to a very high number of predicted rule applications that needs to be reduced again. We found that filtering out all sets of related rule applications for which $|H_{h_{hist},p}| > x$ leads to a better precision.

Using indirect relations allows us to make more complex recommendations, by including more rule applications that are not directly related to the user's changes (G1). This could lead to too specific or complex recommendations that do not fit the user's intentions. We can control this, by adding a parameter that limits the amount of iterations in the search for indirectly related elements (G2). A value of 0 for this parameter turns this extension off, while a value of 1 only allows indirections through one variable (as depicted in the example above), and a value of infinity leads to the search for a fixed point.

3. EVALUATION

In this section, we will describe the data set we used in our evaluation, how we evaluated our technique, and what results we achieved.

CreateEClassInEPackage
↳ selectedEObject: **gmfgraph**
↳ New: **RealFigure**

(a) User added EClass identified by *RealFigure*

AddEClassTgtEClass
↳ selectedEObject: **RealFigure**
↳ NewTarget: `Free1`

(b) Recommended rule application

CreateEAnnotationInEModelElement
↳ selectedEObject: **RealFigure**
↳ New: `Free2`

(c) Recommended rule application

CreateEStringToStringMapEntryInEAnnotation
↳ selectedEObject: `Free2`
↳ New: `Free3`

(d) Recommended rule application

Figure 4: Rule application that describes a user change (4a) and recommended rule applications (4b, 4c, 4d). Based on the historic rule applications in Figure 2

Herrmannsdörfer et al. [8] and Langer et al. [12] have analysed the versions of three different meta models from the Eclipse GMF Project. Kehrer et al. [11] have already applied SiLift to this data set and shown that it can compute correct and complete differences between all versions. The data set consists of the following meta models: (1) `gmfgen` with 110 model versions from 1.139 to 1.248, (2) `gmfgraph` with 10 model versions from 1.23 to 1.33, and (3) `mappings` with 15 model versions from 1.43 to 1.58.

While we use meta model histories in our evaluation since they are readily available, our approach itself is applicable to models as well which we will evaluate in the future.

First, we created asymmetric differences between all versions with SiLift, specifically using its UUID matcher and its atomic rule set. We found the following numbers of Henshin rules per meta model: (1) `gmfgen` 1067, (2) `gmfgraph` 163, and (3) `mappings` 149. Then we extracted the names of the Henshin rules and their arguments from the differences. For every difference $\delta_{i,i+1}$ we used all previous differences $\delta_{j,j+1}, j < i$ as historic differences and the rule applications in $\delta_{i,i+1}$ as the current user changes. That means the changes we wanted to recommend were not part of the historic changes.

We used the current difference $\delta_{i,i+1}$ to validate our recommendations. For every recommended rule application, we tried to find a rule application with the same name and arguments in the current difference. Free variables in the arguments were always counted as the same. A correctly recommended rule application, with correct arguments (or free variables), was counted as one true positive (TP), else it was counted as false positive (FP). Note that the usage of free variables increases the true positive count, but that users also need to do more manual work, which we do not measure here.

Because we do not try to predict the absence of certain rule applications, we can not measure true or false negatives.

We have summarised our results in Table 1 The table shows the number of true (TP) and false positives (FP) per

Name	TP	FP	Precision
gmfgen	554	217	0.72
gmfgraph	25	33	0.43
mappings	36	8	0.82
Total	615	258	-
Average	205	86	-

Table 1: Recommendation results for different models

model. The *Precision* metric is computed by the formula $Precision = \frac{TP}{TP+FP}$ and can be interpreted as the percentage of correctly recommended rules. The results were obtained using all extensions that were discussed in Section 2.3: intersection of recommendations, free variables, and indirect relations. The indirect relations were iterated two times and sets of related historic rule applications with more than five rules were ignored.

Internal validity: Our Evaluation only included three meta models from Eclipse Projects. **External validity:** We have not shown that these models are representative for all meta models or instances of meta models. Furthermore, the batch evaluation preformed in this paper is no substitute for an evaluation with real users as Turpin and Hersh [16] have shown.

4. RELATED WORK

There are many studies and tools about evolving models and meta models, but only a few of them look at model histories and try to automatically recommend changes based on user changes.

Herrmannsdörfer et al. [8] have analysed meta models in the Eclipse GMF Project and developed a set of small change operations that can be used in aggregate to describe all changes to the meta models. Langer et al. [12] developed this idea further by identifying complex change operations that consist of smaller ones. This allows the analysis of a model's evolution on a more abstract level. They have shown that their complex change operations can describe all model changes in the Eclipse GMF Project.

Kehrer et al. [11] have developed a tool that can automatically generate consistency-preserving edit scripts that describe the difference between two versions of the same model. They have implemented the small and complex change operations from [8] and [12] in their tool and evaluated it on the meta models from the Eclipse GMF Project. Their tool could correctly produce edit scripts for all model versions.

In this paper, we have used some of the results from [8], [12], and [11]. Mainly in the form of SiLift for generating our differences and in the form of the Eclipse GMF meta models for our evaluation.

Getir et al. [7] proposed a framework for model co-evolution. Their framework uses model histories, SiLift and Henshin rules to produce suggestions for coupled simultaneous model changes in related instance models. The framework requires manual intervention from developers with domain knowledge. Developers have to add traces between related elements in different models, in order to enable the framework to recognize the relations. The models' version histories and the traces are then used to propose further changes to one model based on user edits in another model. A correlation analysis between change operations in the models' histories is used to predict related change operations. The main difference to our work is that we analyse the arguments of rule application in order to identify related operations. Furthermore, our technique is able to recommend some of the rule application arguments.

Cicchetti et al. [5] present an approach for updating instance models whose meta models have changed. They compute a transformation from the changes in the meta model that can be applied to the instance models so that they conform to the new version of the meta model. This approach does not consider historic model changes and does not try to recommend further changes in the same model based on user changes.

LASE [13] is a tool that can create abstract edit scripts from several similar source code changes made by a user. It matches the change operations in the AST from similar changes, keeping common change operations and abstracting over similar change operations that differ only in variable or method names. The tool can then search for further source code regions that match abstract edit scripts and apply them automatically. This is similar to our technique, although we use historic changes to generate recommendations for a single change made by a user.

Breckel [2] evaluated an approach for automatic error detections in source code. Their approach finds similar but not identical code fragments in a source code file and large code bases. Frequent differences in these code fragments point to bugs in the source code file. This approach can detect typing and more complex errors, is programming language independent, and does not require domain knowledge. This is similar to our technique which automatically compares different versions of models without using domain knowledge about these models.

Bruch et al. [4] developed several systems that incorporate code from repositories to improve the auto completion features of Eclipse. Their systems rank auto completions by finding similar code in the repositories that share variables of the same types. They show that their systems outperform Eclipse code completion. The systems are used to rank auto completions recommended by Eclipse, while our technique generates recommendations based on the change histories of models.

Muşlu et al. [14] developed a technique to improve Eclipse Quick Fix. Their technique automatically counts how many errors are solved or introduced by a quick fix and displays this information to the developer. They improve the source code Editing capabilities of Eclipse by incorporating additional information from the compiler. This technique helps developers to decide between different quick fixes that are already implemented in Eclipse, but does not generate completely new recommendations on its own.

5. CONCLUSIONS AND FUTURE WORK

We have presented our technique for recommending changes based on historical changes. The precision of our recommendations in the evaluation was between 0.43 and 0.82. This shows that our technique could possibly be used to automatically recommend model changes based on user changes and previous model versions.

We plan to improve our technique by addressing the following topics:

- The current aggregation of multiple recommendations via intersections needs to be compared to other tech-

niques, for example from [4], for prioritizing recommendations.

- The recommendations are based on single user changes. Future work should take multiple user changes into consideration.
- Our evaluation only included meta models. We plan to also evaluate our technique for types of instance model for which SiLift can generate differences.
- Our technique needs to be integrated into an intuitive user interface and evaluated by users, because batch evaluations alone are not sufficient [16].
- An extension of our technique that exploits model constraints could filter out recommendations that violate model constraints. It could also be possible to emphasize recommendations that fix constraint violations, which would be similar to Eclipse's Quick Fix recommendations. Muşlu et al. [14] have already done similar work for Eclipse Quick Fixes.
- We want to extend our technique to multiple meta and instance models that are evolving simultaneously. For this, we suspect that it is possible to find relations between changes that are made simultaneously to different models. For example, if two elements are added simultaneously to two different models, we could deduce a relation between them and then try to apply our technique.

6. ACKNOWLEDGEMENTS

This work was funded by the German Research Foundation (DFG) as part of the DFG Priority Programme 1593 (SPP1593).

7. REFERENCES

[1] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[2] A. Breckel. Error mining: bug detection through comparison with large code databases. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 175–178. IEEE Press, 2012.

[3] P. Brosch, M. Seidl, and G. Kappel. A recommender for conflict resolution support in optimistic model versioning. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA*, pages 43–50, 2010.

[4] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[5] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*, pages 222–231. IEEE, 2008.

[6] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[7] S. Getir, M. Rindt, and T. Kehrer. A generic framework for analyzing model co-evolution. In *Model Evolution, International Conference on Model Driven Engineering Languages and Systems*, 2014.

[8] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice: The history of gmf. In *International Conference on Software Language Engineering*, pages 3–22. Springer, 2009.

[9] W. Janjic and C. Atkinson. Utilizing software reuse experience for automated test recommendation. In *8th Int. Workshop on Automation of Software Test*, pages 100–106, 2013.

[10] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. Understanding model evolution through semantically lifting model differences with SiLift. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 638–641. IEEE, 2012.

[11] T. Kehrer, U. Kelter, and G. Taentzer. Consistency-preserving edit scripts in model versioning. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 191–201. IEEE, 2013.

[12] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

[13] 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*, pages 502–511. IEEE Press, 2013.

[14] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. *ACM SIGPLAN Notices*, 47(10):669–682, 2012.

[15] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer, 2014.

[16] A. H. Turpin and W. Hersh. Why batch and user evaluations do not give the same results. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 225–231. ACM, 2001.

On Leveraging UML/OCL for Model Synchronization

Robert Bill
TU Wien, Institute for Software Technology and Interactive Systems
Favoritenstraße 9-11
A-1110 Wien
bill@big.tuwien.ac.at

Martin Gogolla
Universität Bremen,
Department for Mathematics and Computer Science
PO Box 330440
D-28334 Bremen
gogolla@tzi.de

Manuel Wimmer
TU Wien, Institute for Software Technology and Interactive Systems
Favoritenstraße 9-11
A-1110 Wien
wimmer@big.tuwien.ac.at

ABSTRACT

Modelling complex system often results in different but overlapping modelling artifacts which evolve independently. Thus, inconsistencies may arise which lead to unintended effects on the modelled system. To mitigate this situation, model synchronization is seen as a recurring and crucial maintenance task which requires to restore consistency between multiple models using the most suitable changes. Currently, different languages and tools are used for inter-model consistency management than for intra-model consistency where UML/OCL is an accepted solution. Consequently, the result of synchronizing models solely based on inter-model constraints might result into inappropriately evolved models w.r.t. intra-model constraints.

In this paper, we present a synchronization model formalized in UML/OCL which covers explicit consistency and change models including costs and which considers both, inter-model and intra-model constraints at the same time. Instances of this synchronization model represent successful synchronization scenarios. In particular, models can be synchronized, also taking into account their predecessor versions, by finding a constraint violation-free extension of a partial model including those instances which may be optimized for minimal cost. We prototypically implemented this approach using a model finder to automatically retrieve synchronized models and the change operations to compute them by completing the partial model.

Keywords

model consistency; model synchronization; model evolution

1. INTRODUCTION

Modern systems are inherently much more complex to design, develop, and maintain than classical systems due to different properties such as size, heterogeneity, distribution, and multi-disciplinarity. One way to cope with such complexity is by resorting on model-driven engineering approaches [4,5] and dividing the engineering activities according to several areas of concerns or viewpoints, each one focusing on a specific aspect of the system and allowing different stakeholders to observe the system from different perspectives [26, 27]. There are more and more approaches which allow to define different views of a system resulting in partially overlapping models. Unfortunately, this separation of concerns by using different viewpoints, potentially expressed in different domain-specific modeling languages, comes with the price of keeping those viewpoints consistent [13]. Thus, model synchronization has become an indispensable duty where inconsistencies between different models need to be resolved in an efficient and correct way.

There are already several approaches to handle model integration based on synchronization [23]. However, there is a recurring pattern in most of these approaches: a dedicated language is used to define a mix of synchronization steps and inter-model consistency relationships. There are two potential challenges using such approaches: (*i*) for consistency relationship formulation, the intermingling of consistency relationships and synchronization might make it difficult to specify how to realize context-dependent resolutions of inconsistencies in heavily constrained models, and (*ii*) the increased mental load for modelers who need to learn a new synchronization language.

In this paper, we propose a new methodology for unifying inter-model and intra-model constraints for model synchronization using UML/OCL. We employ OCL in the classical setting for defining intra-model constraints and present a method how to define inter-model constraints based on dedicated UML/OCL models between two models in the spirit of bi-directional model transformation languages such as QVT Relations [25] or TGGs [30]. In order to define general-purpose and domain-specific synchronization properties, we introduce a formalized change model that is the basis for explicitly modeling such properties based on our UML/OCL approach. For instance, the use of different cost functions for changes allows the definition and usage of different synchronization strategies such as least-change and beyond. Having these ingredients, UML/OCL based model finders can be employed to compute the model synchronizations taking into account both inter-model and intra-model constraints and the stated properties which should be fulfilled by the synchronization. We show this by applying a model finder for UML/OCL, which has already proven to be usable for transformations models [16].

The remainder of this paper is structured as follows. In Section 2 we describe the architecture of our synchronization approach, the types of models we consider, and the running example of this paper. In Section 3 we formulate model changes as a UML/OCL model and how to select the synchronization strategy by associating changes with cost functions. Subsequently, in Section 4 we describe how inter-model constraints can be expressed using a consistency model based on UML/OCL. In Section 5 we discuss the prototypical implementation of our approach to automatically find model synchronizations. Finally, in Section 6 we discuss related work and conclude with an outlook in Section 7.

2. MODEL SYNCHRONIZATION ARCHITECTURE AND RUNNING EXAMPLE

In this section, we introduce our model synchronization architecture by-example.

2.1 Model Synchronization Architecture

A synchronization problem, as depicted in Figure 1(a) and (b), occurs when two models, which were potentially consistent in their current state, are subsequently changed independently leading to potential inconsistencies. To synchronize both models, the goal is to find a suitable set of changes for both models to make them consistent again. As there may be a huge amount of different change sets to reestablish consistency, the question also arises which one is the most appropriate change set for a given situation. Before we go into details on this aspect, we discuss the general architecture of our approach, in particular, how we represent the model synchronization problem by utilizing change and consistency models.

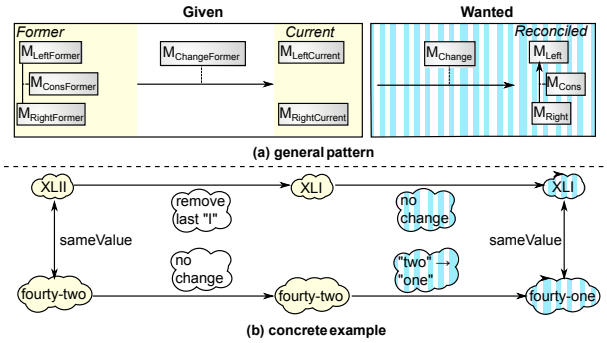


Figure 1: Model synchronizing using change models and consistency models: (a) general pattern and (b) concrete example.

In the following, we will describe the general architecture of our synchronization model as depicted in Figure 1. Our approach synchronizes models by consistently completing a synchronization model containing the history of the models ($M_{\text{LeftFormer}}$, $M_{\text{RightFormer}}$, $M_{\text{LeftCurrent}}$ and $M_{\text{RightCurrent}}$), the previous consistency relation $M_{\text{ConsFormer}}$, which might not be a valid consistency relation in the current state, the previous change set $M_{\text{ChangeFormer}}$, the wanted state of the models M_{Left} and M_{Right} with a consistency relation M_{Cons} and changes M_{Change} leading to a consistent state. In general, model finders like the USE ModelValidator[1] search for instances of a metamodel. A model finder can synchronize models by consistently completing synchronization models consisting of former and current models with consistent change and wanted models by adding new objects, associations and attributes. Since a model finder may complete the model in many ways, including the application of `delete` changes, in the extreme case to delete the complete model, we also foresee to model change costs to guide the model finder in the right direction. In the concrete example shown in Figure 1, the model finder may find and add two change objects (no change; "two" → "one") and two consistent state objects (XLI; forty-one).

Summing up, in contrast to many other approaches, we do not use explicit consistency restoring transformation rules for synchronization, but just the declarative consistency models. Among others, this has the advantage that this approach is, in principle, not limited to synchronize two models, but any number of models, with or without circular consistency dependencies between them. Since both model states are considered at the same time, the typical problem of making model A consistent with model B, thus requiring additional changes in model B which themselves require additional changes in model A etc. can never occur. Also, we do not use any model diffing algorithm, but again a fully declarative change model, i.e., the constraints which have to hold for a model explaining the difference between two model versions. We copy left, right and consistency model two times to build former and current model while removing all constraints, including multiplicity constraints. This is done to ensure that (*i*) the change model actually describes changes between model versions and (*ii*) the model finder can find a consistent model extension defining only the wanted model.

For completeness reasons please note when integrating left models and right models into global models, name clashes may occur. This can be simply avoided by pre- or postfixing names of classes and properties. Since this step is trivial but might clutter the overall architecture and descriptions, it is not further discussed throughout the paper.

2.2 Running Example

As a running example[2], let us consider two viewpoints for developing and maintaining computer networks as depicted in Figure 2: (a) the requirements viewpoint and (*b*) the implementation viewpoint. The metamodels realizing these two viewpoints are illustrated in Figure 3. When taking a closer look on Figure 2, we see that the left model stores the requirements of a computer network. There are named machines which provide a certain amount of communication speed and others which consume a certain expected amount of data. The right part of Figure 2 contains a model of the implemented system which does not only include servers providing data and computers consuming data, but also routers and cables needed to transfer the data from servers to computers. There are two types of cables, namely `GlassFiberCable`s for high-speed connections and `Copper-Cable`s for low-speed connections. There are constraints on the right model to ensure that (*i*) each server can serve the cables it is connected to, (*ii*) each computer gets enough bandwidth for its needs and (*iii*) each router does not produce any data and thus can fulfill the outgoing bandwidth with the incoming bandwidth and its own processing speed.

For the given example, the models' consistency relationship is defined such that each provider needs a server with the same name and at least that speed and each consumer needs a computer with the same name and the same speed and vice versa.

In Figure 2, also the evolution scenario for our running example is shown. It is assumes that the models were changed independently from each other. In the requirements model, the provider `p1` should be made ready for the future and gets a higher speed. Thus, its speed attribute was increased from 3 to 4. At the same time in the implementation model changes were performed. It was discovered that the computer `w2` was never used and thus it was removed together

[1] `https://sourceforge.net/projects/useocl/files/Plugins/ModelValidator/`

[2] A slightly simplified version of the example can be downloaded from `http://cosimo.big.tuwien.ac.at/findsync/`

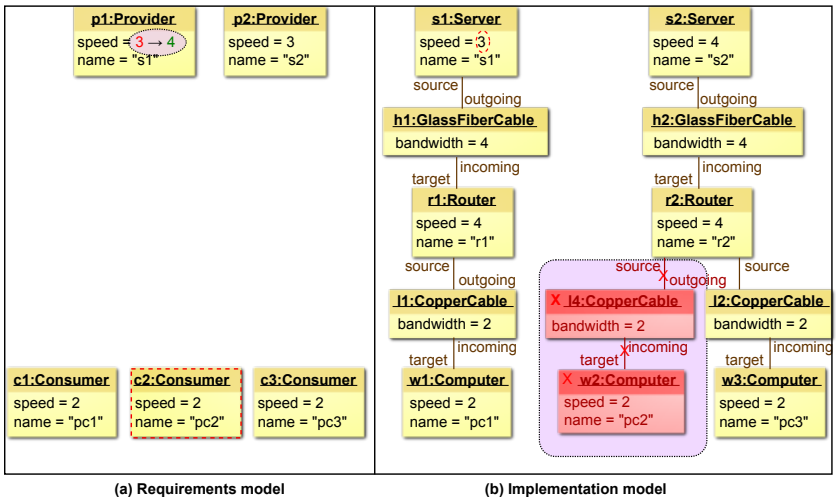


Figure 2: Example synchronization scenario: a requirements model, its corresponding implementation model, and their uncoordinated evolution.

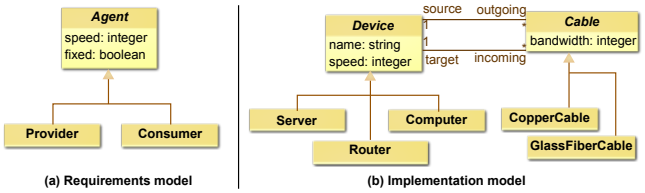


Figure 3: Metamodels of the running example: (a) requirements metamodel and (b) implementation metamodel.

with its connected cable `l4`. Now there are two violations w.r.t. the aforementioned consistency relationship: The speed of server `s1` is not sufficient and the consumer `c2` has no correspondence on the right side. Of course, one now may come up with some reconciliation actions for this small example. One obvious model synchronization would be to delete consumer `c2` in the requirements model and to set the speed variable to 4 for server `s1`. For larger examples, of course automation support is needed in order to reason about appropriate synchronizations. How this is realized by our approach following the previously described architecture is the content of the following sections.

3. CHANGE MODEL

Our change model approach consists of an abstract part which is the same for all modeling languages and a language-specific part which is generated for each used metamodel individually. In general, we follow the ideas presented in [9] to generate language specific change models to represent changes between two models. However, we also go beyond the ideas presented in [9], by providing also the conditions for finding a valid change model. Thus, we do not only explicitly model the abstract syntax of change models but also explicate their semantics in terms of OCL constraints.

3.1 Abstract Change Types

Figure 4 shows the abstract structure of our change model. There are two types of changes, namely atomic changes and composite changes [21]. An atomic change connects the original object to the revised object. If an object has been deleted, there is no revised object. If an object has been created, there is no original object. If an object is preserved, there are both original and revised objects. In the change model, every original object must specify what happens in the future and every revised object must specify what happened to it in the past. For set-valued features, the set of future values must be equal to the set of past values with the deleted values removed and the created features added. For bag features, the number of occurrences of a future feature must be equal to the occurrence count of this feature in the past object plus the sum of all added feature counts minus the sum of all deleted feature counts. For ordered features, the sequence resulting from the deletion of all deleted elements from the past object feature must be the same as the sequence resulting from deleting all created elements from the future object feature which is expressed by inserting values at specific list indexes, sorted from bottom to top, to the common base sequence.

A composite change builds higher level changes from lower level changes [21]. For example, type changes cannot be directly represented with atomic changes. A general cast combines an atomic delete change and an atomic create change to express that semantically, the object has not been deleted and another created, but the object is still the same. Similarly, a general move combines an atomic feature delete change and an atomic feature create change for the same feature and the same value to specify that the feature has moved and was not deleted and re-added. A feature change is a composite change defined by an OCL operation which takes several parameters and has a postcondition defining which changes are done to the object. The last change type resembles composite changes as proposed in literature

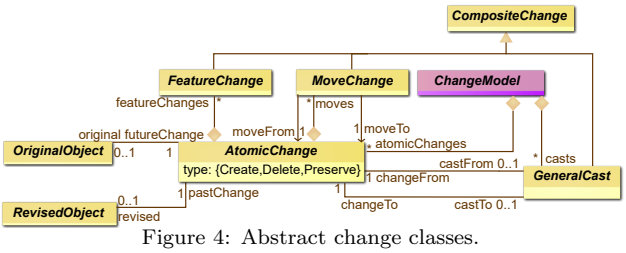


Figure 4: Abstract change classes.

(e.g., [19]) while the others could also be semantically regarded as atomic changes, but are regarded as composite changes from an implementation point of view. In principle, such explained changes could be derived (semi-)automatically for various postconditions [24]. If such technologies are used, only the cost remains to be specified.

Currently, these are the only composite changes supported, but others might be added in the future. Also, it is currently not possible to let a feature change explain features of multiple objects.

3.2 Language-specific Change Language

Beside the change model itself, the metamodel is duplicated twice as well where class and feature names are changed to avoid name clashes. In particular, all model classes and associations of the previous version might be suffixed with `Prev` while the metamodel to store current objects and feature values is suffixed with `Ori` and names do not change for the synchronized model.

Two parallel class hierarchies resembling the class hierarchy of the original metamodel are created. In the `Change`-hierarchy, all classes are replicated with a Change prefixed name inheriting from `AtomicChange`. For every structural feature in the original class *X*, two structural features of the same type *T* named as the original structural feature *attr* plus a suffix of `Add` and `Del` are created in the `Change`-class as shown in List. 1. They denote which values have been added or deleted by the change operation. The `Add` attribute is derived as values which exist in the revised model, but not the original model and the `Del` attribute is derived as values which exist in the original model, but not the revised model. Unique features generate a Set type, non-unique features generate a Bag type. The derived attributes *attr*`AddCostly` and *attr*`DelCostly` contain unexplained addition and deletion changes. Associations are derived in a similar fashion. To ensure that added and deleted classes can be derived via set difference, the change objects have to be used instead of the original objects, i.e. `revised.`*attr*`.futureChange`/`original.`*attr*`.pastChange` is used instead of `revised.`*attr*/`original.`*attr*.

List. 1: Change calculation for unordered features.

```
context ChangeX:
 attrAdd: [Set|Bag](T) derived = if revised =
     null then [Set|Bag]{} else revised.attr->
     asBag() endif - if original = null then [Set
     |Bag]{} else original.attr->asBag() endif
 attrDel: [Set|Bag](T) derived = if original =
     null then [Set|Bag]{} else original.attr->
     asBag() endif - if revised = null then [Set|
     Bag]{} else revised.attr->asBag() endif
 attrAddExpl: [Set|Bag](T) derived =
     featureChanges.attrAddExpl->union(
     castChanges.attrExpl)->union(movedFrom.
     attrExpl)
 attrDelExpl: [Set|Bag](T) derived =
     featureChanges.attrDelExpl->union(
     castChanges.attrExpl)->union(movedTo.
     attrExpl)
 attrAddCostly: [Set|Bag](T) derived = attrAdd -
     attrAddExpl
 attrDelCostly: [Set|Bag](T) derived = attrDel -
     attrDelExpl
```

For every ordered structural feature, three additional features named as the original structural feature plus `AddEl`, `DelEl` and `Same` are created in the `Change` class as shown in List. 2. The feature `*Same` contains a base feature value which is extended by inserting the values in `*AddEl` to get the revised feature value and extended by inserting the values in `*DelEl` to get the original feature value. The feature `*Same` has the same type as the original feature, but `*AddEl` and `*DelEl` are sequences of pairs of `(Integer,`*T*`)`. List. 2 shows the connection between all features.

List. 2: Additional change calculation for ordered features.

```
context ChangeX:
  attrAddEl: Sequence(Tuple{i: Integer, v: \myvar
      {T}})
  attrDelEl: Sequence(Tuple{i: Integer, v: \myvar
      {T}})
  attrSame: Sequence(T)
  attrMoved: Bag(T) derived = attrAddEl.v -
      attrAdd
  attrMovedExpl: Bag(T) derived = featureChanges.
      attrMovedExpl->union(castChanges.
      attrMovedExpl)

  inv orderedSequence: Set{attrAddEl,attrAddEl}->
      forAll(s | s->isUnique(i) and s->sortedBy(i
      ) = s)
  inv addToRevised: (revised = null) or (revised.
      attr = attrAdd->iterate(t, full =
      <@attrSame | full->insertAt(t.i, t.v)))
  inv delFromOriginal: (original = null) or (
      original.attr = attrDel->iterate(t, full =
      <@attrSame | full->insertAt(t.i, t.v)))
```

The sequence `*Same` is an auxiliary sequence that represents a common sequence between both ordered features. The feature *attr*`AddEl` contains not only elements which have been added to the revised feature from the original feature, but also elements which are not in the common sequence because they have changed their position. Elements which are moved are those which are not in the common sequence but whose value has not been added to the common bag as well. The constraints guarantee a certain order of insert applications and ensure that the original and revised feature value actually can be built as previously described. Unlike the change object above, these feature values are not necessarily directly determined by original and revised objects. In fact, the sequence in the *attr*`Same` feature might be too short; then too many moves are determined. However, a larger number of moves yields higher costs and thus a solution with less superfluous moves will be preferred.

Each `Change` object also has a `original` and `revised` association of the corresponding classes in the original and the revised model which redefines the `original` and `revised` association in `AtomicChange` to ensure the correct type.

Consider Figure 5 as example for an excerpt of the change model for atomic changes of our running example. Since `Device` contains a name attribute, `ChangeDevice` contains two attributes to define which names have been added and deleted. Also, the `incoming` and `outgoing` associations are used to connect to added and deleted cables. The class `ChangeRouter` adds two attributes for the `maxSpeed`. It does not replicate the `name` attribute changes since they are available in the superclass.

Let us now consider another excerpt of an instance of the change model as depicted in Figure 6. There are two original objects, the original cable `oc1` and the original router `or1`. The cable has been deleted, so the speed attribute is deleted as well as the incoming connection. The router

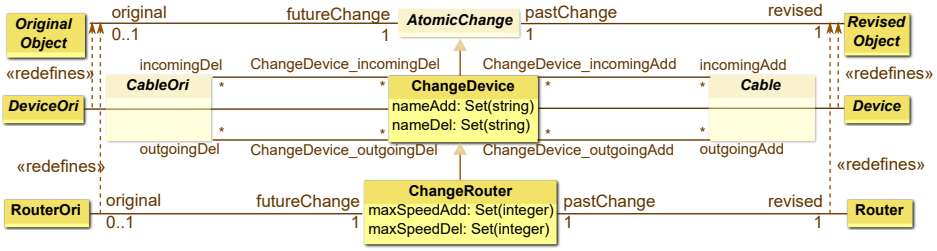


Figure 5: Change model excerpt 1 of the running example.

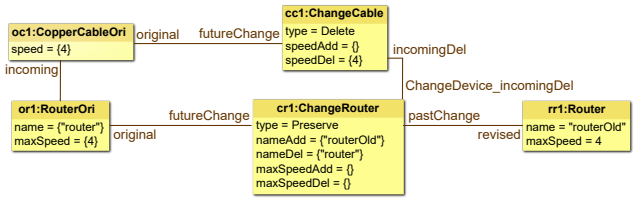


Figure 6: Change model excerpt 2 of the running example.

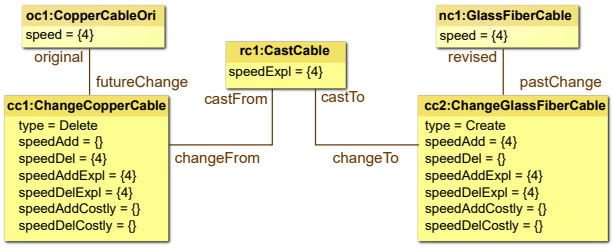


Figure 7: Cast change excerpt for the running example.

just has been taken out of usage, so its name was changed to *routerOld* from *router*. In terms of attribute changes, this equals the addition of *routerOld* to the name and the deletion of *router*. The speed has not been changed, so no additions or deletions are defined there. The fact that all constraints, including multiplicity constraints, are removed from all but the target model is indicated by the set-valued name and speed attributes for the original objects.

Composite changes.

In the `Cast` class hierarchy similarities between cast objects are stored. For every structural feature in the original class, a structural feature of the same type named as the original structural feature plus a suffix of `Expl` is created in the `Cast` class. It denotes which values are the same in both objects and is defined by the intersection of values between original and revised object and thus explains those ostensible changes. List. 3 demonstrates the structure of `Cast` objects. An invariant ensures that the cast object is connected to a single object which is deleted and an object which is added.

List. 3: Change calculation for casts

```
context CastX:
 attrExpl: [Set|Bag](T) derived = castFrom.
     attrDel->intersection(castTo.attrAdd)
 inv changeTypes: changeFrom.type = ChangeType.
     Delete and changeTo.type = ChangeType.Create
```

Consider Figure 7 for an example of such a cast change for our running example. The model has only changed by retyping `oo1` from `CopperCable` to `GlassFiberCable`. This is expressed as two changes, namely deleting the CopperCable and creating a GlassFiberCable with the same values. Since the speed has not changed, the same value was deleted for the CopperCable that was added to the GlassFiberCable. Thus, the cast explains this value change and it is not considered for costs.

Similarly, a `Move` change for a certain aggregation attribute connects two change objects where an association to a certain object *X* was deleted in one object, but created in another object. However, there is no requirement that any object must be deleted or created.

List. 4: Change calculation for moves

```
context MoveX:
 attrExpl: [Set|Bag](T) derived = moveFrom.
     attrDel->intersection(moveTo.attrAdd)
```

In the example depicted in Figure 8, an incoming edge was moved from the Router `or1` to `or2`. The two router change objects show that the edge was deleted (`cr1`) and added (`cr2`). The move object `m1` connects both changes. The intersection of `incomingAdd` for `cr2` and `incomingDel` of `cr1` is `cc1`, so this is the attribute change explained by the move object. Thus, it is also explained in both `cr1` and `cr2` and no change of `incoming` is a `costly` change.

For every operation *O* in a class *X*, a corresponding `FeatureX_O` class is created which inherits from `FeatureX`. This class contains an attribute for each operation parameter and it has a bidirectional association to the change class of the class the operation is defined in. The postcondition of the operation *X* is converted into an invariant of the generated class by changing the `self` context of each `@pre` expression to the `original` object of the associated change while other `self` contexts are transferred to the `revised` object of the associated change. For features changed in the referenced object, this class contains a `T[Add|Del]Expl` attribute for each attribute in the original class. Postconditions which have the pattern `feature = feature@pre->including([expr])` or `feature = feature@pre->excluding([expr])` are converted into the invariant `T[Add|Del]Expl = Set{[expr]}`.

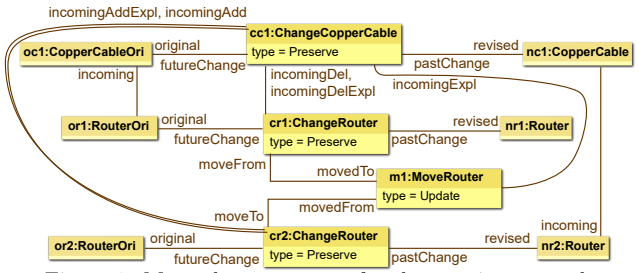


Figure 8: Move change excerpt for the running example.

Additionally, invariants are generated to ensure all features not occurring in any such postcondition are empty. An invariant is generated to let the cost attribute of the class be equal the operation result.

List. 5: Simple change operation defined in UML/OCL.

```
context Router::disconnectServer(c: Cable)
  post disconnectedIncoming: incoming = (
       incoming@pre)->excluding(c)
  post disconnectedOutgoing: outgoing = (
       outgoing@pre)->excluding(c)
```

This example shows a simple operation which just disconnects a single cable from a router. The postconditions state that after this operation has been called, the disconnected cable is not connected to the server any more.

List. 6: Translation of the change operation of List. 5 into the synchronization model

```
class FeatureRouter
attributes
  ...
end

class FeatureRouter_disconnectServer <
    FeatureRouter

constraints
 inv disconnectedIncoming:  incomingDelExpl = Set
     {c}
 inv disconnectedOutgoing:  outgoingDelExpl = Set
     {c}
 inv noChange: incomingAddExpl = Set{} and ...
end

association
    assoc_FeatureRouter_disconnectServer_c
    between
  FeatureRouter_disconnectServer[*] role
      FeatureRouter_disconnectServer_c
  ChangeCable[1] role c
end
```

This example is translated as shown in List. 6. For each postcondition with the specified pattern, an invariant is generated which explains the according changes. The third invariant guarantees that all other attributes are not changed due to this operation call. The operation parameter is translated as association. Like for atomic changes, a parameter of type Cable is translated to a ChangeCable.

3.3 Change Costs

While the change model describes what has changed, for synchronization purposes, the severity of changes should be known. To describe this aspect as well, a cost function is added to each `Change`-class. The specification of the exact cost function depends on the synchronization strategy. In we following, we discuss how different synchronization strategies can be implemented with our approach.

Least change.

The least change [22] strategy cost function sums up the size of all unexplained add and deletion collections in the `Change` objects and adds 1 if the change type is `Delete` or `Create`. If objects have been casted, one change should not be counted. This function can be easily changed to support a **least creation** or **least deletion** strategy by adding a higher value instead of 1 to `Delete` or `Create` changes. Positive costs ensure hippocraticness [32], i.e., that the synchronization does nothing for consistent models, because every change would have positive cost while consistent models would require no change which has no and thus less cost.

List. 7: Least change cost calculation.

```
class ChangeObject
attributes
  costObject: Integer derived = if type =
       ChangeType.Create or type = ChangeType.
       Delete then if castTo != null then 0 else 1
        endif else 0 endif
end

class ChangeCable
attributes
  costCable: Integer derived = speedAddCostly->
       size() + speedDelCostly->size() +
       sourceAddCostly->size() + sourceDelCostly->
       size() + targetAddCostly->size() +
       targetDelCostly->size()
end

class CostSummer
attributes
  cost = ChangeObject.allInstances().costObject->
       sum() + ChangeCable.allInstances().
       costObject->sum() + ...
end
```

List. 7 shows an excerpt of the implementation of the least cost strategy for the running example. The cost of deleting or adding objects is defined as attribute of the `ChangeObject` class. If an object is casted to another object, the cost is not accounted. The `ChangeCable` class calculates the cost by summing up the costs of its direct attributes. The `CostSummer` calculates the total costs by summing over all cost attributes.

Real costs.

Models might be representations of the real world where changes induce real costs, e.g., working time for switching cables or the costs of a new server. Then, it is natural to use a cost function resembling real costs. These costs will typically occur as operation costs.

List. 8: Domain-specific cost calculation.

```
context Router::disconnectServer(c: Cable)
  body: if c.isKindOf(CopperCable) then 5 else 10
        endif
==>
class FeatureRouter_disconnectServer
attributes
  cost: Integer derived = if (c.revised.isKindOf(
       CopperCable)) then 5 else 10 endif ...
```

List. 8 shows a simple example of operation costs. It might be more difficult to disconnect glass fiber cables than to disconnect copper cables, so the costs could be twice as high. Such costs are transformed into an invariant as explained in the previous section.

Organizational asymmetry.

If model A is considered more important than another model B, changes in model A should be avoided. By assigning much higher costs to changes in the model A, changes in

model A are avoided but enforced in model B. For instance, this strategy may be also of relevance for the running example of this paper.

Avoiding undos.

Usually, a model was modified for a good reason. Thus, the synchronization step should not return to the previous model version. To achieve that, additional costs can be introduced if some attributes in the target re-appear: (*i*) if they were deleted before or vice versa using the sum of symmetric difference sizes of `attrAdd` and `attrDelete` of future and past change objects in the current model, (*ii*) if objects are deleted when they were created before by counting create/delete pairs, and (*iii*) if objects are created when they were deleted by counting created objects in the target model which are similar to deleted objects in the former model.

List. 9 shows a simple example for avoiding undos. Each attribute which is reassigned to the same value as before will induce a cost of 10 because both the old value is added which should not be the case and the newly set value is deleted. If an attribute whose value was changed has its value changed again the cost would only be 5. If an object is deleted which was created before, the cost increases by 100. Likewise, if an object is recreated which has the same name as an object which was deleted, the cost is increased by 200.

List. 9: Costs for avoiding undos.

```
class ChangeRouter
attributes
  avoidRedoCosts: Integer derived = if original.
      pastChange.type = ChangeType.Create then 5*
      original.pastChange.speedDel->intersection(
      speedAdd)->union(original.pastChange.
      speedAdd->intersection(speedDel))->size()
      +...
  else if type = ChangeType.Delete then 100 else
      0 endif endif + if type = ChangeType.Create
       and ChangeRouterOri.allInstances()->select
      (c | c.type = ChangeType.Delete).original->
      select(r | r.name = revised.name) then 200
      else 0 endif
end
```

Retaining existing traces.

A consistency model might be just descriptive or prescriptive. In the former case, changes in the consistency model should have no cost, while in the latter case, consistency model changes, especially deletions, might have high costs if the associated objects were not deleted.

Please note that there are design decisions to take which synchronization properties to define as constraints and which via costs. For example, undos could not only be avoided using costs as described in List. 9, but also by just using a constraint like in List. 10. However, then no synchronization would be possible at all if conflicting model changes would have been performed which could only be resolved by undoing a change. This also holds in the case of asymmetric changes. If a model should never change, this model can just be used as target models with additional constraints that all attributes and associations should remain equal.

List. 10: Constraints for avoiding undos.

```
class ChangeRouter
constraints
  inv avoidRedo: if original.pastChange.type =
      ChangeType.Create then type <> ChangeType.
      Delete and if original.pastChange.type =
      ChangeType.Delete then type <> ChangeType.
      Create endif
end
```

4. CONSISTENCY MODEL

In this section, we will show how inter-model consistency concerns are modeled with UML/OCL. In general, we follow the main idea of triple graph grammars (TGGs) [30] and triple patterns [10] to build an explicitly modeled structure between two models. If the models are defined with UML/OCL we end up with an unified representation for intra- and inter-model concerns.

As we now show in this section, by using UML/OCL it is possible to describe the interconnection between models by additional associations, constraints and possibly even additional other elements such as classes and attributes.

Since we aim for a non-intrusive addition of synchronization logic, we assume open models or some form of module import, at least for defining new associations. In the following, `@override` denotes the extension of the specified class with attributes and associations contained by the consistency model. We now show several examples how to employ UML/OCL to define consistency models in terms of correspondences.

One-to-one correspondences.

One-to-one correspondences are easily defined in UML/OCL by adding an association between the corresponding classes having as lower and upper bounds 1 on both ends. Figure 9 shows an example where classes correspond based on name equality. Such constraints can either be added to a class (as shown in graphical syntax) or added to the association (as shown in textual syntax) as preferred.

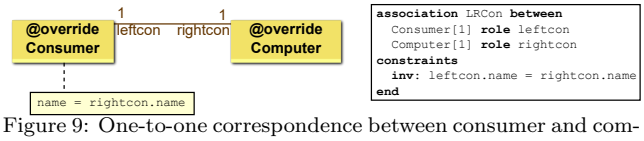


Figure 9: One-to-one correspondence between consumer and computer in graphical and textual syntax.

One-to-many correspondences.

One-to-many correspondences can also be expressed by associations. Figure 10 shows an example where a server in the requirements model might correspond to a whole cluster in the implementation model. In particular, the constraint `inv1` specifies that a single cluster provider corresponds to many nodes if all servers have exactly one router and nothing else as target of their outgoing cables. Then, all servers connected to this router constitute the cluster, else a provider corresponds to a single cluster.

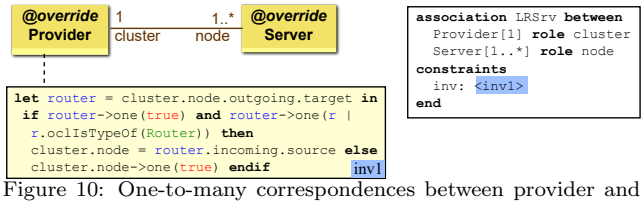


Figure 10: One-to-many correspondences between provider and server in graphical and textual syntax.

Many-to-many correspondences.

Many-to-many can also be expressed either by using n-ary associations of UML or by introducing additional classes which connect more than one class on both sides by associations. Figure 11 shows a more complex correspondence between provider and server. Servers may be either part of a typical cluster where a service is provided by multiple servers or part of a virtual cluster, where services may be assigned to many servers and each server might handle multiple services.

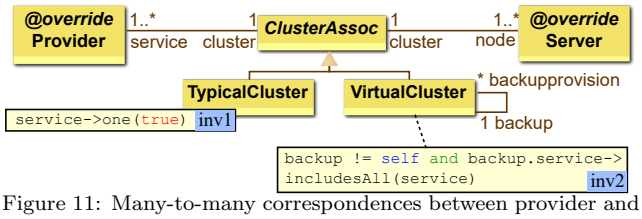


Figure 11: Many-to-many correspondences between provider and server in graphical syntax.

Correspondence dependencies.

A full consistency model may also contain dependencies between correspondences modeled as associations. Figure 11 shows that each virtual cluster requires a second virtual cluster as backup which must provide the all services.

5. TOOL SUPPORT BASED ON THE USE MODEL VALIDATOR

The presented synchronization approach has been prototypically implemented as an USE plugin and can be downloaded from `http://cosimo.big.tuwien.ac.at/findsync`. While the consistency model has to be hand-crafted, the USE plugin merges the different involved models automatically, generates the change model, and finds the minimum cost synchronized models. To give an idea about the complexity of the automatically produced models, consider the running example of this paper for which the generated model contains 79 classes, 124 associations, and 379 invariants.

The costs are currently fixed to a constant for each primitive change operation. Due to limitations in the USE model validator, some constraints and structures had to be reformulated and only set-typed features are supported. The optimization process is run by iteratively finding a model completion with less cost than the previously found solution. If the model finder is not able to find any better solution, the cost-optimal solution has been found. Bounds have to be given for the number of instances of each class, thus cost-optimality is only guaranteed with regards to these bounds.

5.1 Evaluation

We ran our performance evaluation using a simplified synchronization scenario of the one shown in Figure 2 on a Intel i5-6500 3.2 GHz machine with 64 GB RAM, running Ubuntu Linux 16.04. For simplicity reasons, all costs were assumed to be one.

[3]Version '15, available at: `http://fmv.jku.at/lingeling/lingeling-bal-2293bef-151109.tar.gz`

[4]Version 2.2.0, available at: `http://minisat.se/downloads/minisat-2.2.0.tar.gz`

[5]Version 2.3.1, integrated in the ModelValidator jar, available at: `http://forge.ow2.org/project/download.php?group_id=228&file_id=17186`

Solver	Time (FS)	Costs (FS)	Time (OS)
lingeling[3]	4 min 52 sec	21	38 min
plingeling[3]	6 min 26 sec	17	77 min
MiniSat[4]	8 min 49 sec	46	OOM
Sat4J[5]	65 min	3	85 min

Table 1: Execution times and costs for the running example (FS: first solution, OS: optimal solution, the cost for producing the optimal solution is 2, OOM: out of memory).

Table 1 shows the average runtime of three runs to find any solution and a cost-minimal solution with different solvers. The runtimes vary greatly with each run, but still some trends can be observed. The runtimes also indicate that with the current state of the USE model validator, the approach cannot be directly used to find a good synchronized instances within a reasonable time frame for large models. Still, we see a huge difference between solvers. While lingeling and plingeling are quite fast finding the first solution which already provide a good quality, Sat4J requires a longer time to find any solution, but this first solution is nearly the optimal one. MiniSat had quite some problems for the studied scenario. It produced did not produce appropriate solutions and went out of memory without finding the best solution.

In addition to computing synchronizations, we see also an alternative usage scenario for the presented approach. It can be also used to validate whether an existing synchronization found by another tool is a valid synchronization by using the validation capabilities of OCL. Here, the full approach can be used since USE is able to check all constraints used in the approach.

5.2 Threats to Validity

In the evaluation, we have only shown the general feasibility of the approach, but not that it is fast enough for larger instances. We assume that faster, maybe heuristic model finding approaches may significantly improve the performance but we do not have any concrete evidence for this yet. Thus, we can not claim that the performance can be actually improved in order that the approach scales to the synchronization of larger real-world models in the matter of minutes.

6. RELATED WORK

While transformation models using UML/OCL have been already introduced back in 2006 [3], to the best to our knowledge, the model synchronization aspect has not been considered in such type of models. However, since model synchronization is an important topic in model-driven engineering, there is already a great variety of approaches to support different synchronization scenarios. For example, there is specific work on model synchronization by specifying how to deal with inconsistencies [2, 11, 14] and by using bidirectional transformation languages. For instance, triple graph grammars [20, 30] are often employed for model synchronization scenarios such as reported in [1, 15, 17]. In a similar fashion, QVT Relational [33] allows synchronization, also in conjunction with unidirectional transformation languages such as ATL [22]. Furthermore, there are other, mostly rule-based, approaches available (cf. [18] for a survey). One benefit of our approach when comparing it to existing work

is that we make use of an explicit change model which allows to adjust the synchronization strategy in a domain-specific way.

There are also some existing approaches using constraint solving for model synchronization such as the Janus Transformation Language (JTL) [8], also with compressed state space [12], and the CARE approach [29]. JTL [8] is using answer set programming (ASP) to find a synchronized result. In contrast to JTL, we do not map a relation-based language into ASP, but we aim to describe everything with UML/OCL. In addition, we also provide a method to prefer certain change results over others by having cost models attached to change models. In previous work, we have presented CARE [29], an approach using a constraint solver (ASP) to re-synchronize models with their evolving metamodels. However, CARE is a specific approach for the metamodel/model co-evolution problem. The approach presented in this paper may be also employed in the future to reproduce the results of the CARE approach. An alternative approach for metamodel/model co-evolution is presented in [28] where the variability of different model migration solutions is formalized as a feature model. We see this research direction as an interesting line for future work as this would allow to concisely report equally good model synchronization solutions, i.e., having the same cost, to the user.

There have been many change models proposed in the literature, e.g., [6, 9, 31, 34]. However, to the best of our knowledge, we do not know any constraint-based approach to define change models.

7. CONCLUSION AND FUTURE WORK

In this paper, we have shown how to transform the problem of model synchronization into the problem of defining a suitable consistency model and a change model with UML/OCL. This can be used to check whether a given synchronization strategy was performed successfully and to find synchronization strategies by model completion. As the evaluation has shown, the approach can be used to build an incremental transformation which is used for our approach itself. The approach has also been prototypically implemented which shows that automation is feasible.

In the future, we need to further develop both the conceptual approach and the implementation. Sometimes, you explicitly require a change in a model if a specific change happened in another model [35]. Within this approach, this would correspond to a consistency model constraining change objects. A sensible consistency model or an alternative representation translated into a consistency model for that has to be found. The approach could be adapted to support metamodel/model co-evolution by (*i*) translating the metamodel to a model a consistency relation between this generated model and the model to co-evolve or (*ii*) allowing the target metamodel to differ from the source metamodel. The first approach would be more general since it would not only cover model evolutions based on metamodel changes but may also cover metamodel evolutions based on model changes, but at the same time more challenging since OCL constraints would have to be translated into a model and interpreted using OCL. Thus, we are currently evaluating the latter approach.

The performance and scalability of the implementation has to be assessed in order to increase its practical usefulness. Including optimization strategies in the USE model validator for this problem, which has been done in the past for similar problems, may increase the performance by orders of magnitude and make this approach usable in practice. Also, we need to evaluate in which cases, if any, a suitable cost definition could let our approach find *least surprising* changes [7].

8. ACKNOWLEDGMENTS

Acknowledgment: This work has been funded by the Vienna Business Agency (Austria) within the COSIMO project (grant number 967327) and by the Christian Doppler Forschungsgesellschaft, the Federal Ministry of Economy, Family and Youth and the National Foundation for Research, Technology and Development, Austria.

9. REFERENCES

[1] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr. Efficient Model Synchronization with View Triple Graph Grammars. In *Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA)*, volume 8569 of *LNCS*, pages 1–17. Springer, 2014.

[2] G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations - change (in) the rule to rule the change. *SoSyM*, 11(3):431–461, 2012.

[3] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 4199 of *LNCS*, pages 440–453. Springer, 2006.

[4] J. Bézivin, R. F. Paige, U. Aßmann, B. Rumpe, and D. C. Schmidt. Manifesto - model engineering for complex systems. *CoRR*, abs/1409.6591, 2014.

[5] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[6] E. J. Burger. Flexible views for view-based model-driven development. In *Proceedings of the 18th International Doctoral Symposium on Components and Architecture*, pages 25–30. ACM, 2013.

[7] J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. Towards a principle of least surprise for bidirectional transformations. In *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015.*, pages 66–80, 2015.

[8] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: A Bidirectional and Change Propagating Transformation Language. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, volume 6563 of *LNCS*, pages 183–202. Springer, 2011.

[9] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[10] J. de Lara, E. Guerra, and P. Bottoni. Triple patterns: Compact specifications for the generation of

operational triple graph grammar rules. *ECEASST*, 6, 2007.

[11] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo. Change management in multi-viewpoint system using ASP. In *Workshops Proceedings of the International IEEE Enterprise Distributed Object Computing Conference (EDOCW)*, pages 433–440. IEEE, 2008.

[12] R. Eramo, A. Pierantonio, and G. Rosa. Managing uncertainty in bidirectional model transformations. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 49–58, 2015.

[13] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In *Proceedings of the 4th European Software Engineering Conference (ESEC)*, volume 717 of *LNCS*, pages 84–99. Springer, 1993.

[14] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Trans. Software Eng.*, 20(8):569–578, 1994.

[15] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *SoSyM*, 8(1):21–43, 2009.

[16] M. Gogolla, L. Hamann, and F. Hilken. On static and dynamic analysis of UML and OCL transformation models. In *Proceedings of the Workshop on Analysis of Model Transformations co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014)*, pages 24–33, 2014.

[17] F. Hermann, H. Ehrig, C. Ermel, and F. Orejas. Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 7212 of *LNCS*, pages 178–193. Springer, 2012.

[18] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. Feature-based classification of bidirectional transformation approaches. *SoSyM*, 15(3):907–928, 2016.

[19] M. Javed, Y. M. Abgaz, and C. Paul. Composite ontology change operators and their customizable evolution strategies. In *Proceedings of the 2nd Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn)*, pages 1–12. CEUR-WS.org, 2012.

[20] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, tr-ri-07-284, University of Paderborn, 2007.

[21] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

[22] N. Macedo and A. Cunha. Least-change bidirectional model transformation with QVT-R and ATL. *SoSyM*, pages 1–28, 2014.

[23] N. Macedo, J. Tiago, and A. Cunha. A feature-based classification of model repair approaches. *CoRR*, abs/1504.03947, 2015.

[24] P. Niemann, F. Hilken, M. Gogolla, and R. Wille. Assisted generation of frame conditions for formal models. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 309–312, 2015.

[25] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. `http://www.omg.org/spec/QVT/1.0/`.

[26] J. E. Rivera, J. R. Romero, and A. Vallecillo. Behavior, time and viewpoint consistency: Three challenges for MDE. In *Models in Software Engineering, Reports and Revised Selected Papers of Workshops and Symposia at MODELS 2008*, pages 60–65, 2008.

[27] J. R. Romero and A. Vallecillo. Well-formed rules for viewpoint correspondences specification. In *Workshops Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (ECOCW)*, pages 441–443, 2008.

[28] D. D. Ruscio, J. Etzlstorfer, L. Iovino, A. Pierantonio, and W. Schwinger. Supporting variability exploration and resolution during model migration. In *Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA)*, volume 9764 of *LNCS*, pages 231–246. Springer, 2016.

[29] J. Schoenboeck, A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Schwinger, M. Wimmer, and M. Wischenbart. CARE: A Constraint-based Approach for Re-Establishing Conformance Relationships. In *Proceedings of the 10th Asia-Pacific Conference on Conceptual Modelling (APCCM)*, pages 19–28. Australian Computer Society, 2014.

[30] A. Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 151–163, 1994.

[31] C. Seidl, I. Schaefer, and U. Aßmann. Deltaecore - A model-based delta language generation framework. In *Proceedings of Modellierung*, pages 81–96, 2014.

[32] P. Stevens. Towards an algebraic theory of bidirectional transformations. In *Proceedings of the 4th International Conference on Graph Transformations (ICGT)*, pages 1–17, 2008.

[33] P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *SoSyM*, 9(1):7–20, 2009.

[34] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *SoSyM*, 13(1):239–272, 2014.

[35] M. Wimmer, N. Moreno, and A. Vallecillo. Viewpoint co-evolution through coarse-grained changes and coupled transformations. In *Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS)*, volume 7304 of *LNCS*, pages 336–352. Springer, 2012.

Challenges in the Evolution of Metamodels: Smells and Anti-Patterns of a Historically-Grown Metamodel

Misha Strittmatter
Chair for Software Design and Quality (SDQ)
Karlsruhe Institute of Technology
Karlsruhe, Germany
strittmatter@kit.edu

Georg Hinkel
Software Engineering Division
FZI Research Center of Information Technologies
Karlsruhe, Germany
hinkel@fzi.de

Michael Langhammer
Chair for Software Design and Quality (SDQ)
Karlsruhe Institute of Technology
Karlsruhe, Germany
langhammer@kit.edu

Reiner Jung
Software Engineering Group
Christian-Albrechts-University Kiel
Kiel, Germany
reiner.jung@email.uni-kiel.de

Robert Heinrich
Chair for Software Design and Quality (SDQ)
Karlsruhe Institute of Technology
Karlsruhe, Germany
heinrich@kit.edu

ABSTRACT

In model-driven engineering, modeling languages are developed to serve as basis for system design, simulation and code generation. Like any software artifact, modeling languages evolve over time. If, however, the metamodel that defines the language is badly designed, the effort needed for its maintenance is unnecessarily increased. In this paper, we present bad smells and anti-patterns that we discovered in a thorough metamodel review of the Palladio Component Model (PCM). The PCM is a good representative for big and old metamodels that have grown over time. Thus, these results are meaningful, as they reflect the types of smells that accumulate in such metamodels over time. Related work deals mainly with automatically detectable bad smells, anti-patterns and defects. However, there are smells and anti-patterns, which cannot be detected automatically. They should not be neglected. Thus, in this paper, we focus on both: automatically and non-automatically detectable smells.

Keywords

metamodel smells; metamodel maintainability; metamodel anti-patterns; Palladio Component Model

1. INTRODUCTION

Model-driven engineering uses domain-specific modeling languages (DSMLs) to express abstractions of reappearing domain concepts. DSMLs can then be used to design systems. The resulting models (instances of DSMLs) can be analyzed and used for simulation. Depending on the DSML, its instances can be used for purposes ranging from stub generation to the generation of fully functional code. DSMLs are defined by metamodels.

The evolution of metamodels is a big challenge, as they tend to be central artifacts with many tools that depend on them. A change in a metamodel may cause many errors in dependent code. Thus, it is important that metamodels allow the addition of new features by creating metamodel extensions (which we will refer to as *external extensibility*). Some bad smells in metamodels make external extensions impossible. Other bad smells degrade a metamodels general maintainability.

In this paper, we present the results of a thorough metamodel review of the PCM in the form of a list of bad smells. The PCM [26] is a metamodel to describe component-based software architectures with a focus on their performance properties. It is implemented in EMF's Ecore [29], which is an implementation of EMOF [24]. The metamodel is approximately 10 years old and has been extended by features over time. While later features have been extended externally, earlier additions were made intrusively. Thus, the PCM is a representative specimen of a big, old and grown over time metamodel. That is the reason why we think the results of our review are important to the community. They show what type of smells typically accumulate in metamodels over time.

Related work mainly focuses on the automatic detection of metamodel smells and simple defects of the metamodel. Being able to detect these problems automatically is important. However, smells and anti-patterns that cannot be automatically detected should not be ignored. Thus, besides four bad smells that can be automatically detected, we also present 6 smells which can only be detected manually. This paper does not cover any metamodel defects, which break conformance to the meta-metamodel (comparable to compiler errors in programming). These defects are usually taken care of by the modeling frameworks validation functionality.

This paper is structured as follows: Section 2 gives foundational information about metamodels, their evolution and maintainability, anti-patterns and bad smells. Section 3 presents the state of the art in evaluating metamodel quality and detecting bad smells in metamodels. Section 4 will briefly give background information and present the internal structure of the PCM. Section 4 will show how the PCM has

developed over the year. Section 5 contains the list of bad smells in the PCM. Section 6 concludes the paper.

2. FOUNDATIONS

In this paper, we are concerned with Essential Meta-Object Facility (EMOF)-conforming metamodels [24] (e.g. instances of the EMF's Ecore meta-metamodel). A *metamodel* defines and constrains the set of its instances (i.e. models). In the sense of EMOF, a metamodel consists of metaclasses, which in turn contain relations and attributes. If the elements of a model conform to the definitions in the metamodel, the model is an instance of the metamodel. EMOF-conforming metamodels are similar to UML class diagrams. The differences are that they have to be complete and have to form a containment tree.

The relations that can be defined in a metamodel connect two metaclasses. Attributes of metaclasses have only primitive types. Metaclasses are able to inherit from each other. A special case of a relation is the containment relation. Each element of a model, except its roots, has to be contained in another element. A metamodel can be subdivided into packages. A metamodel may also reference metaclasses of another metamodels. Constraints can be defined (e.g. using the Object Constraint Language (OCL) [25]).

Metamodels evolve because their languages evolve. New features have to be added, concepts have to be adapted, and bugs have to be fixed. A more detailed classification of metamodel evolution can be found in [31]. In our experience with the PCM, the biggest driver is the inclusion of new features.

How easy it is to evolve a metamodel can be considered the *maintainability* of a metamodel. The maintainability is influenced directly by a metamodel's complexity and understandability. Understandability is not completely derived from complexity, as it is possible to metamodel a simple concept in a way which is not intuitive. The concepts of cohesion [36] and coupling [4] can be transferred from object-oriented software development. The *cohesion* of a module is described as how related its classes are. *Coupling* of one module to another expresses how dependent the first module is on the second. Both measures are heuristics for maintainability. A high cohesion is beneficial, as related classes tend to evolve together. A high coupling between packages is detrimental, because modifications may have a bigger impact on dependents.

In object-oriented software development, a *bad smell* [9] is considered an indicator for a possible problem in the software's design or code. An anti-pattern was originally defined as being "... just like pattern, except that instead of a solution it gives something that looks superficially like a solution, but is none." [27]. However, the meaning of *anti-patterns* changed over time to mean a recurring pattern that has negative consequences [28, 14], regardless if it was purposely used or not. When transferring these terms to the domain of metamodeling, some bad smells can be defined as anti-patterns [2]. Other bad smells may be indicated by metrics [2]. Some are only detectable by manual investigation. In the following, we will refer to bad smells in metamodels as *metamodel smells* or simply *smells*. Metamodel smells may have various negative effects on metamodel maintainability, which we will explain in this paper. In our list of smells, we include automatically detectable smells, but even more importantly smells which can only be reliably detected manually. As the PCM is a valid metamodel, metamodeling errors (which prevent the generation of the model code) will not appear in our list. For each smell we identified, we explain the characteristics of this smell, its consequences, reasons why they appeared in the PCM, how we think they can be best corrected, whether we can automatically detect them and where they occurred in PCM.

3. STATE OF THE ART

EMF Refactor [2, 1] is a tool that can be used to automatically detect bad smells and perform refactorings in Ecore-based metamodels and UML models. They detect bad smells either by a violation of a specific metric or by the detection of an anti-pattern. For Ecore they feature automated detection of the following anti-patterns[1]: Large EClass, Speculative Generality EClass, Unnamed EClass. They feature an even longer list for UML anti-patterns. Of these, some may also be applicable to Ecore metamodels.

Elaasar [7, 8] developed an approach for automated detection of patterns and anti-pattern in MOF-based models. His approach provides a ready to use catalog with patterns specifications but also supports the creation of new pattern specifications by the user. His MOF anti-patterns are grouped in the categories well-formedness, semantic and convention.

López et al. propose a tool and language to check for properties of metamodels [20]. In their paper, they also provide a catalog of properties, which they categorize in: design flaws, best practices, naming conventions and metrics. They check for breaches of fixed thresholds for the following metrics: number of attributes per class, degree of fan-in and -out, depth of inheritance tree and the number of direct subclasses.

Vépa et al. present a repository for metamodels, models, and transformations [35]. The authors apply metrics that were originally designed for class diagrams onto metamodels from the repository. For some of the metrics, Vépa et al. provide a rationale how they relate to metamodel quality.

Di Rocco et al. [6] applied metrics onto a large set of metamodels. Besides the usual size metrics, they also feature the number of isolated metaclasses and the number of concrete immediately featureless metaclasses. Further, they searched for correlations of the metrics among each other. E.g., they found that the number of metaclasses with superclass is positively correlated with number of metaclasses without features. Based on the characteristics they draw conclusions about general characteristics of metamodels. Their long-term goal is to draw conclusions from metamodel characteristics concerning the impact onto tools and transformations that are based on the metamodel.

Gómez et al. [13] propose an approach, which aims at evaluating the correctness and expressiveness of a metamodel. I.e. weather it allows invalid instances (correctness) and is it able to express all instances it is supposed to (expressiveness). Their approach automatically generates a (preferably small) set of instances to evaluate these two criteria.

García et al. [11] developed a set of domain specific metamodel quality metrics for multi-agent systems modeling languages. They propose three metrics: availability, specificity and expressiveness. These metrics take domain knowledge into account, e.g., the "number of necessary concepts" or the "number of model elements necessary for modeling the

[1]https://www.eclipse.org/emf-refactor/index.php

system of the problem domain".

There is much work on quality metrics for object-oriented design and UML class diagrams [5, 22, 21, 12]. Further, there are publications that present empirical analyses of object-oriented design metrics [3, 34]. E.g. Subramanyam found that the correlation between metrics and bug detection varied when applied to different programming languages and observed interactions between metrics. The purpose and usage of object-oriented design and class diagrams is very different compared to metamodels, thus their benefit cannot be assumed for metamodels.

4. THE PALLADIO COMPONENT MODEL

The Palladio approach [26] is an approach to component-based software engineering. At its core is the PCM, a metamodel, which defines a language to express component-based software architectures and abstractions of several quality aspects. In this section, we present insights into the structure of the PCM.

The PCM is separated in different hierarchical packages. The root package is called `pcm` and directly or indirectly contains the remaining packages. In this paper, we consider packages, that are directly contained in the root packages as first level packages. Packages contained in first level packages are considered as second level packages and so on. The containment hierarchy of the packages is depicted in Figure 1, while dependencies between the packages implied by inheritance of classes within the packages are depicted in Figure 5.

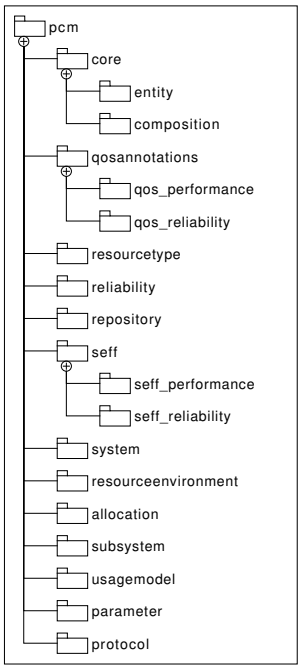


Figure 1: The package containment hierarchy of the PCM

In [32], we identified different main concerns of the PCM, which can be seen in Figure 2. The figure shows dependencies how they should be, not how they actually are. As one can see, in these two figures, the packages are mainly sliced as the concerns. For instance, the `repository` package contains all classes that are necessary to build a `Repository` with its `DataTypes` and `ComponentTypeHierarchy`. However, some packages contain information for multiple concerns.

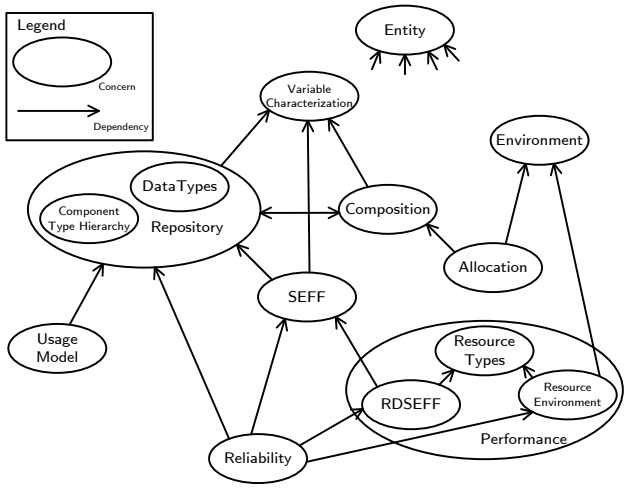


Figure 2: The main concerns of the PCM [32]

Starting in August 2006, the PCM has a long evolution history. Since then, many new features have been added. Although Palladio initially was built for performance prediction, it now also supports to simulate reliability, data consistency, energy consumption and maintainability. At the same time, besides the original call characteristics, a system can now be specified in an event-based manner and PCM has built-in support for extensions in the form of stereotypes.

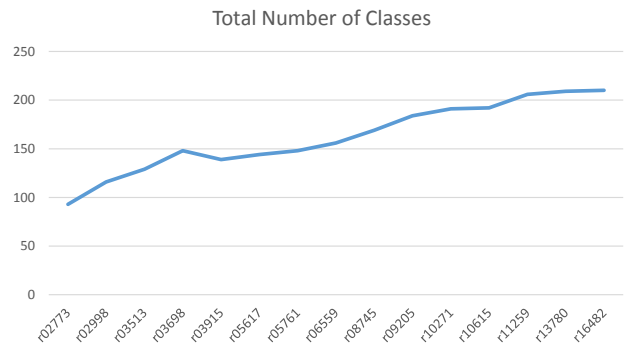


Figure 3: Evolution of the Palladio Component Model between March 2007 and September 2012 in terms of the number of classes

Many of these perfective changes have had a footprint to the metamodel. Today, the version control system registers more than 120 revisions of the core metamodel plus several revisions for underlying shared functionality it is using. To give an impression on the evolution of the metamodel, we have depicted the total number of classes between spring 2007 and fall 2012 in Figure 3. Most changes to the PCM were mad in this period and the metamodel size in terms of number of classes has more than doubled.

5. METAMODEL SMELLS

In this Section, we will present the list of metamodel smells we found in the most recent version of the PCM. This version of the metamodel was released with the Palladio Bench 4.0. For each metamodel smell, the following aspects will be elaborated: general description, its consequences, reasons for its forming, or even rationale why it might have been purposefully used, possible resolutions, if the smell is automatically detectable as well as brief mention of its occurrences in the PCM. In a smell's description, we will also briefly discuss the relation of a smell to object-oriented design and programming (OO).

5.1 Redundant Container Relation

Description: Containment relations are necessary to be able to specify entities that are more complex and they guide serialization. To navigate models, it can be necessary to traverse from a contained element to the containing element. For this purpose, the container reference can be used. In EMF, this feature is provided by a generic and implicit reference eContainer. However, it is also possible to define an explicit container reference utilizing the concept of opposite references.

This smell does not exist in OO, as there are in general no explicit containers. Objects are contained in the heap memory and are merely referenced by other objects. If no more references to an object exist, the object is eventually deleted. In metamodeling and modeling on the other hand, an EObject must have a container. If that container is deleted, all contained elements are also deleted and all references to the deleted elements are unset.

Consequence: This smell has several negative implications. First, it introduces redundancy, as the implicit eContainer reference is always present and an explicit container reference is a duplication. Second, it increases metamodel complexity due to this duplication. In case a class is used in different containments, multiple explicit container references exist. These references are all mutually exclusive, but this cannot be declared with EMF itself. Furthermore, the opposite references must be declared as optional, which weakens their meaning. For example, the `PCMRandomVariable` of the PCM metamodel is used in 17 contexts. Therefore, the class has 17 opposite references, but only one is used by an instance. Third, the explicit container reference can harm reuse and evolution of metamodels. In case a container class is added or removed, this always also requires the adaption of the contained class. In case different aspects and partitions of metamodels are stored separately, a cyclic reference between the metamodel of the containing class and the contained class is necessary. This hinders reuse, as both metamodels must be present. However, using an implicit eContainer reference, the metamodel with the contained class can be reused in other contexts.

Reason and Rationale: Some may argue that container references allow ensuring static type safety. However, this only applies for cases with only one container reference and realized in Java directly. In case of multiple containing classes, the static type safety property is weakened, as the containing class type cannot be determined statically. Instead, at runtime each property must be checked which is similar to testing the type of the containing class, but it is obfuscated that this is indeed a type check.

In some UML metamodels, associations are used where both ends are named, and one end is declared as composite. If this property of the UML metamodel must be preserved, an explicit container reference cannot be omitted. However, such naming can exist only for documentary reasons, which allows ignoring them for the EMF mapping of the UML metamodel.

Correction: The explicit container reference can be removed and its usage in code can be replaced by accessing the eContainer reference. If only the explicit container reference was used before, the eContainer reference can be safely cast to that container. If, however, the explicit container reference is checked for null, the eContainer reference has to be checked for the type of the expected container.

Automatic Detection: Opposite references for containment references can be detected automatically.

Occurrences: In the PCM, 85 explicit container references can be found with 17 of these references originating from the aforementioned `PCMRandomVariable`.

5.2 Obligatory Container Relation

Description: This smell is a special case of the redundant container relation smell (Section 5.1). If a containment relation has an opposite reference that has a lower bound of 1, we call it an obligatory container relation. This is illustrated in Figure 4. Class `C1` contains `A` and the container relation is obligatory. As with the redundant container relation smell, this smell is not relevant in OO, as there is no explicit containment.

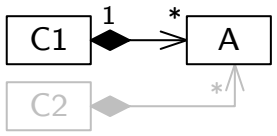


Figure 4: Obligatory Container Relation for Classes C1 and A

Consequence: `A` cannot be used in any other context. E.g., although `C2` has a containment relation to `A`, an instance of `C2` can never contain any instances of `A`, as the container relation to an instance of `C1` has to be set. In such cases, the EMF framework does not even allow code generation.

Reason and Rationale: There are some reasons to use an obligatory container relation. It ensures type safety when navigating to the container. It is also possible, that the developers want to restrict reuse explicitly. However, in most circumstances the developers were most likely unaware of these consequences. Obligatory container relations can also be the result of translation from another format or language (e.g., UML) by a transformation.

Correction: To fix this smell, remove the container relation (this will also resolve the redundant container smell). As the class could only be in one type of container, the eContainer can be safely cast to that container class.

Automatic Detection: Obligatory container relations can easily be detected. However, manual evaluation is still required, as in some cases they may be intended by the developer.

Occurrences: In the PCM, many container relations are obligatory. We suspect this to be the result of the transformation from Rational Software Architect.

5.3 Concern Scattered in Package Hierarchy

Description: The package hierarchy of a metamodel is mainly for logical partitioning of its content. We consider it a bad smell, if the classes that constitute a feature or concern of the language are spread over multiple packages. Even crosscutting concerns can be modularized in a more meaningful way.

In OO, there are issues similar to this smell, e.g., when cohesive classes are scattered over packages or assemblies. However, to our knowledge, there is no explicit smell that covers the problem on this level. OO smells are more concerned with the internals of packages: relations between classes and the internals of classes and methods.

Consequence: This bad smell has negative consequences on the understandability and thus maintainability of metamodels. When a developer tries to understand a metamodel, he examines its packages and from their content and documentation (if there is any) tries to conclude its purpose. If a concern is scattered, the purpose of the package cannot be fully comprehended without tracing relations that leave the package. The smell may also increase coupling between affected packages and reduce relative cohesion within the packages.

Reason and Rationale: We suspect that this smell occurs mostly, when new concerns are implemented in an already existent metamodel. The new concern is related to the concerns of multiple other packages. Parts of the new concern are then placed in the packages of the related concerns and so the new concern is ripped apart.

Correction: A better approach would be to place the new concern in its own package. The package should further contain sub-packages for each related concern, which then contain the classes that are related to these concerns. The package of the new concern should be placed meaningfully. If it is a first order concern, it should be placed below the root package. If it is a subconcern, its package should be placed as a subpackage of the parent concern. If it is a crosscutting concern, it should be placed on the same level, as the concerns it is intersecting with.

Moving classes can be done through refactorings. Even the code, which depends on the classes, may be automatically fixed. A mere moving of affected classes may lead to other bad smells, if the dependencies are not modified. The new dependencies between packages may lead to package dependency cycles (see Section 5.5) and violations of the dependency inversion principle on the package level (see Section 5.6). This is not the fault of consolidating a concern, but of dependencies that were improper in the first place. An explicit reference structure (e.g., [33, 30]) can help in structuring packages and directing their properties properly.

Automatic Detection: This bad smell is not automatically detectable. An algorithm is not able to automatically infer the semantics of parts of the metamodel.

Occurrences: It is difficult to nail down the exact number of occurrences in the PCM, as this depends on how fine-grained its concerns are identified. Looking at quite coarse-grained concerns, there are at least six occurrences of this smell in the PCM [32]. The following concerns are affected: resource interfaces, middleware infrastructure, performance, repository (especially interfaces), event communication and reliability.

5.4 Multiple Concerns in Package

Description: Conversely to the scattered concern smell, we also consider it a smell, if a package contains the classes of multiple concerns. The relation of this smell to OO is analog to the scattered concern smell. Insufficient modularization on the package level is an issue in OO. However, we are not aware of an explicit smell definition.

Consequence: Having multiple concerns in one package, increases the effort to understand the package, because the developer has to identify the contained concerns and their respective classes. Simply put, the package is needlessly complex. This bad smell might also decrease the cohesion within the package.

Reason and Rationale: We suspect that this smell has two explanations. Developers tend to place classes in packages, which hold their container or represent a closely related concern. It is just more convenient to use the existing package hierarchy than to think of a new structure yourself.

Correction: How to modularize and package concerns is already well explained in the resolution part for the scattered concern smell (see Section 5.3). As already suggested, new concerns should be placed in their own package. If a concern is a subconcern, then its package should be placed as a subpackage.

Automatic Detection: This bad smell is not automatically detectable. An algorithm is not able to automatically infer the semantics of parts of the metamodel.

Occurrences: There are at least two occurrences of this smell in the PCM [32]. The following concerns are affected: data types and the abstract component type hierarchy, which both are located in the repository package.

5.5 Package Dependency Cycles

Description: When creating a metaclass, one of the most important choices is the selection of appropriate base classes, from which some functionality can be reused. However, inheritance is a white-box technique. Therefore, one needs to fully understand the base classes. Hence, a closer look into the containing packages is required. This dependency between packages implied by inheritance can be shown visualized as a graph. We depicted this graph for the latest PCM version in Figure 5. This graph may contain cycles (shown in red).

In OO, having dependency cycles in assemblies is consid-

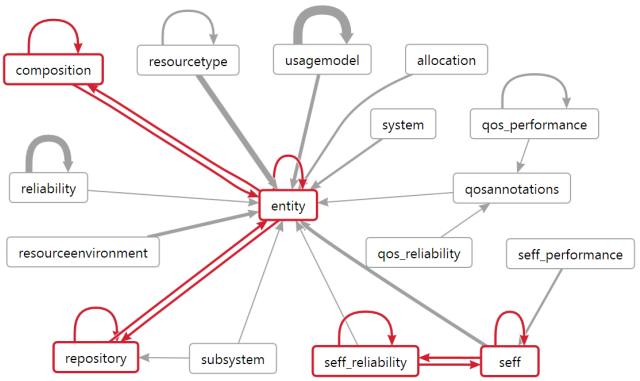


Figure 5: Dependencies between packages in PCM implied by inheritance of classes, thickness of arrows indicates how many classes use an inheritance relation, cycles are marked in red.

ered a bad practice, or is even treated as an error on some platforms. However, there is not so much emphasis placed on dependency cycles on the package level.

Consequence: A consequence of such a circular dependency may be that to fully understand a package contained in such a circle, a developer has to understand all packages contained in this circle. This challenges the appropriateness of the package structure.

Especially if the cycle is formed from inheritance dependencies, the maintainability of the metamodel may suffer. Changes made to the metamodel propagate down the class hierarchy and thus into other packages where they should not.

Reason and Rationale: PCM makes extensive use of multiple inheritance and the inheritance hierarchy of some metaclasses is quite high. Therefore, it may have become difficult to keep an eye on package dependencies.

Correction: In situations when developers have lost an overview of the package dependencies, we think that an overview such as in Figure 5 can already be helpful to avoid this anti-pattern.

Automatic Detection: A circular dependency can be detected automatically (in fact, Figure 5 is entirely generated by a tool) and could be even automatically resolved by merging the affected packages. However, we think a manual inspection can be more beneficial in such a scenario.

Occurrences: The occurrences of this pattern in the latest version of Palladio is shown in Figure 5.

5.6 Dependency Inversion Principle Violated

Description: The dependency inversion principle [23] is a design principle from OO. When translated to metamodeling it states that high-level classes should not depend on low-level classes (high- and low-level regarding the level of abstraction). Both may depend on abstractions. The same can be said about packages and even metamodels, when the dependencies of a package or metamodel are regarded as the combined dependencies of their elements.

Part a) of Figure 6 shows a violation of the principle on the class level. Package H contains high-level concepts, relative to which the content from the package L is low-level. Class A has a dependency (relation, inheritance or containment) to K. Thus, a high-level class is dependent on a low-level one. Class K contains further information about A (indicated by the data attribute). Although it is illustrated as a single attribute, this information may come in the form of attributes, relations and containments.

Consequence: A violation of the dependency inversion principle may have detrimental effects on the maintainability of a metamodel. During evolution, modifications of a concern may influence a more high-level concern. Such violations do also hinder understanding. When a developer tries to understand a concern, he may trace the outgoing relations to more low-level concerns. Thus, he may examine concerns which are not necessary for understanding the high-level concern or even irrelevant to his intent.

Reason and Rationale: In our opinion, these violations stem from the integration of features. It is most convenient for a developer to extend an existing class hierarchy by adding dependencies that point to the new content. However, there is a difference between object-oriented design and metamodels. In OO, it is easier and more natural to introduce new abstraction layers. In metamodels, on the other hand, interfaces cannot be used in a similar way, because usually it is not similar functionality that is added, but new and different data. At first glance, it seems to be a good solution just to create a new subclass, which adds the needed information. However, when multiple new features are implemented this way, they cannot be combined. Therefore, in metamodeling, it can be reasonable to violate the dependency inversion principle in certain cases. A possible rule would be to do so for core features of the language. A feature can be considered a core feature, if it is useful in every use case of the language and for every possible type of user. Core features should be integrated intrusively into the metamodel with a violation of dependency inversion principle. This has the following advantages: type safety, adherence to cardinality and retrieval in $O(1)$ (constant time).

Correction: When implementing non-core features, the dependency inversion principle should be used. In Figure 6, we point out two possible options.

In b), the new abstract class B is created as well as a con-

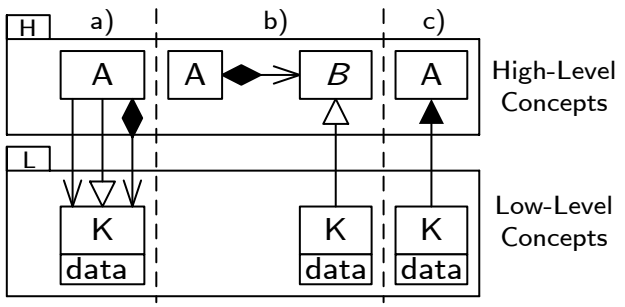


Figure 6: Violation and Application of the Dependency Inversion Principle

tainment from A to B. Class K then inherits from B. Thus, the dependency is reversed and now goes from L to H. This solution has some benefits. The instances of K are contained in instances of A. This enables direct navigation and thus retrieval in constant time. In addition, the cardinality can be controlled directly without having to specify complex constraints. However, type safety is not guaranteed, as the extended data is not placed in B but in K. This solution has to be enabled in the initial development, as the class B is required. This is no issue, if K is also already created during the initial development or if the future extension of K can be foreseen. The main disadvantage of this solution is that H has to be modified, if this solution should be implemented in hindsight.

In c), an alternate solution is shown which does not require modification of H. By either stereotype application [18, 17], aspect-oriented extension [15, 16] or plain referencing, instances of K can be associated with instances of A. Compared to b) this has the disadvantage, that the look-up is in $O(k)$, where k is the number of instances of K.

Automatic Detection: This smell is not automatically detectable. An algorithm is not able to deduce if concepts are higher- or lower-leveled compared to others.

Occurrences: Within the PCM is at least one serious occurrence of this smell with regard to the inheritance hierarchy. The superclass that represents entities that require and provide interfaces, inherits from a superclass which represents entities which require and provide resource interfaces. While the usage of interfaces is a core feature of the PCM, resource interfaces are not. Regarding violations with references, there are countless occurrences. They stem from the intrusive integration of features into more abstract concepts.

5.7 Dead Class

Description: As a result of a refactoring, in some cases a class is no longer required as its responsibilities are taken care of by another class. Sometimes, although the references to the class are deleted, the class itself is not. In OO, this smell falls under the category of dead code or oxbow code.

Consequence: This has a negative impact on understandability since the class has to be considered, even though it cannot be contained in the rest of the model. Furthermore, developers may have a hard time trying to understand how the class is used. When they finally find out that it is not used at all, this has an impact on their opinion of the metamodel.

Reason and Rationale: The pattern is mainly a result of bad metamodel reuse. As the metamodel gets large, it is no longer obvious in which places a class is used. Thus, when the class is no longer required in one specific scenario, it still may be required in another. However, as developers do not check whether the class is used in some other place, the class may be left behind with no usage elsewhere in the metamodel.

Correction: This problem can be avoided if developers make sure that classes they no longer need are either still used elsewhere or deleted.

Automatic Detection: It is possible to statically detect that a class cannot be contained in another class. However, a manual assessment is then required to decide whether this class is dead, as for root container classes, it is viable (though not obligatory) to be uncontainable.

Occurrences: A static analysis of PCM delivers in total nine uncontainable classes. From these, the classes `UsageModel`, `Repository`, `ResourceRepository`, `System`, `ResourceEnvironment` and `Allocation` represent view types and therefore serve as model roots. Thus, it is perfectly valid for them to be uncontainable. The class `DummyClass` has been introduced to overcome a technical limitation in the QVT-O compiler, but this is a rather different issue (the purpose of this class is hardly documented, but developers trying to understand PCM would not expect any reasonable semantics from a class named like this). However, over the history, two classes have been left over from refactoring operations, `CharacterisedVariable` and `ResourceInterfaceProvidingRequiringEntity`. For both of these cases, it is not obvious that they are no longer needed, so developers may try to find usages and fail to do so.

5.8 Concrete Abstract Class

Description: This smell is concerned with classes that should be abstract, but are not. Usually, in a class hierarchy, a class with subtypes is abstract. However, not every occurrence is necessarily bad design, as sometimes even a concrete class might have concrete subclasses.

In OO, having a concrete abstract class is also a problem. However, we are not aware of an existing smell definition.

Consequence: However, if a class that should be abstract is not declared as such, this has a negative impact on the metamodels correctness and understandability. Due to the fact, that an instance of the metamodel may validly contain direct instances of a class that should not have any instances, the metamodel is less correct. Usually this problem is hidden by self-built model editors, which just do not offer any possibility to create direct instances of the affected class. However, using fully generated model editors (like the EMF tree editors), this problem does manifest. Further, the understandability of the metamodel is slightly reduced by this smell. A developer, who investigates the metamodel, cannot instantly identify the class as abstract and has to reflect.

Reason and Rationale: We expect this smell to appear mainly because of carelessness mistakes.

Correction: The correction of this smell is trivial. The affected class just has to be declared as abstract.

Automatic Detection: Occurrences of this smell can only partly be detected automatically. When a concrete class has subclasses, it might be a true case of this smell [2]. If any of the subclasses are abstract, it is even more likely that there is an issue. However, manual evaluation is still required, as it might be the case that the superclass is validly concrete. In constellations, where all subclasses of the concrete abstract class are in external metamodels, the smell is not detectable if the external metamodels are not analyzed. This smell might lead to wrongful detections of the dead class smell

(5.7). This is the case when the class and its superclasses are never used within the metamodel but carry the information of an abstract concept that should be specialized in an external metamodel.

Occurrences: Within the PCM, the `CallReturnAction` class from the `SEFF` package is a true occurrence of the concrete abstract class smell. It is a concrete superclass and cannot be instantiated by the custom build graphical model editors of the PCM bench. This class cannot be meaningfully instantiated, as it or its superclasses have no container. However, it can still be confusing for a developer.

5.9 Duplicate Features in Sibling Classes

Description: In metamodels, classes represent concepts and inheritance is used to specialize concepts by providing a more comprehensive specification. For example, an abstract component type only describes that a component type has an interface. This class can be specialized into a component type that allows having internal components. In case a class has multiple children, of which some realize the same feature, this can be seen as a redundancy in the model.

In OO, one could argue that this issue falls under the duplicate code smell [9].

Consequence: Redundant declarations harm maintainability, as they must be maintained equally in all classes. If one class is overlooked, the metamodel degrades, which hinders long time evolution of the metamodel. They also have a negative impact on implementing transformations, as for each sibling class, the transformation rule must be able to support the feature. This is necessary, as from a syntactic viewpoint on the model these features are different.

Reason and Rationale: Duplicated features can appear when metamodels are altered iteratively. In that case, one of the sibling classes is extended with a specific feature, and later another sibling is extended in the same way. Through this process, more and more classes have a semantically identical feature, but they are declared syntactically as different features. While in some cases this situation may be the effect of limited time, carelessness, or overlooked, it can also be made intentional. The latter case occurs when not all siblings require the feature.

Correction: To mitigate the issue of duplicate features, in OO, a refactoring would move the feature up to the parent class in case all siblings have the feature [19]. However, this can be in violation with the underlying semantics of the concepts, which are expressed in the classes. Furthermore, the pull up cannot be used in cases where not all, but some siblings declare the same feature. An alternative strategy is to define an interface that provides the feature in question and inherit the interface by all siblings that require the particular feature. The strategy has two advantages. First, the meaning of the feature is encapsulated in its own concept. Second, the interface can be used in transformations. Therefore, the transformation must only test whether the interface exists instead of testing multiple classes.

Automatic Detection: An automatic detection of duplications based on name and type is unreliable, as the detection is based only on syntactic properties. Therefore, the detection may result false positives and false negatives. First, there could be identical typed and named features that do not represent the same relationship. Second, features may be named differently, but still represent the same idea. Therefore, manual intervention is required.

Occurrences: In the PCM, the classes `OperationSignature`, `InfrastructureSignature` and `EventType` have all a property `returnType` with the same intended semantic. These could be extracted into an `IValueReturning` interface, which is inherited by the three classes.

5.10 Classification by Enum

Description: If an enum is used as an alternate way to classify a class, we consider it a bad smell. This should not be confused with using an enum to model a mere property.

Using an enum for classification is one possible solution of how to model multiple orthogonal classifications. Part a) of Figure 7 illustrates the problem. It should be possible to classify the class Base as either A1 or A2 and additionally either as B1 or B2. This is not possible by just using an inheritance hierarchy of the depth of just one. Part b) of Figure 7 shows a solution by using an enum for the second classification dimension.

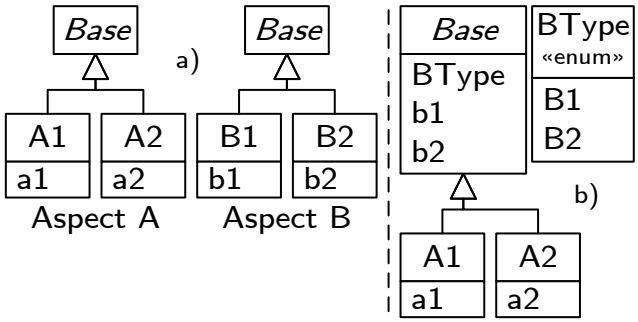


Figure 7: a) Problem of Orthogonal Classifications b) Classification by Enum

The naive solution to modeling orthogonal classifications is shown in Figure 8. There, every possible combination is explicitly modeled by inheritance. This obviously has several disadvantages. It produces high amounts of classes. Although, a single classification dimension is externally extensible, it is not possible to develop independent extensions, as every combination of every dimension has to be modeled.

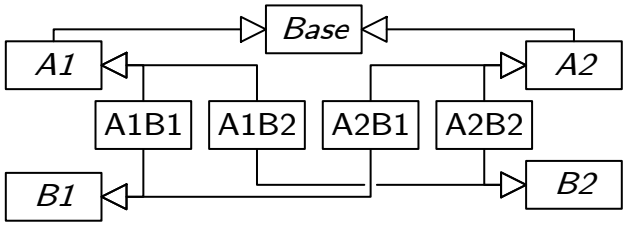


Figure 8: Naive Solution to Orthogonal Classifications

In OO, classification by enum is also a problematic solution to the problem of orthogonal classification dimensions.

However, we are not aware of any bad smell definition.

Consequence: Using an enum for additional classifications makes the classification impossible to extend externally. Further, in contrast to classification by inheritance, it is not possible to add features to parts of the classification selectively. This might lead to the developer adding features to the base class, which are only used for specific values of the enum. By doing that, the complexity of the class increases unnecessarily and its understandability suffers. This is shown in part `b)` of Figure 7.

Reason and Rationale: As already stated, using an enum for classification is one possible solution of how to model multiple orthogonal classifications. In most situations, however, it is not a very suited one. Developers use it because of lack of knowledge of more appropriate solutions. In addition, it looks simple and little intrusive compared to the naive approach (see Figure 8). If, however, the developer wants one classification to be closed for extension and it does not carry any new features that vary for its subtypes, classification by enum can be legitimately used.

Correction: There are two ways to resolve this smell. If the classification is already known when the metamodel is initially implemented, or it is possible to modify the metamodel, the *composition over inheritance principle* [10] should be applied. This is shown in part `a)` of Figure 9. If not, stereotypization should be applied. This is shown in part `b)` of Figure 9.

Automatic Detection: This smell is not automatically detectable. One could scan for each usage of an enum. However, not every usage is a classification. Therefore, each enum usage has to be checked manually.

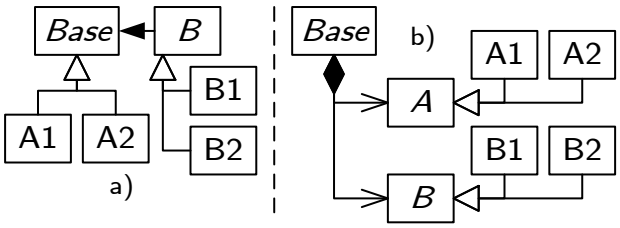


Figure 9: Solutions to Orthogonal Classifications

Occurrences: In the PCM, the classification by enum occurs in the class `ImplementationComponentType`. It contains an enum that declares its component type: business component or infrastructure component. This classification is orthogonal to the classification of atomic vs. composite, which is implemented by inheritance. This enum makes it impossible to add further component types without intrusively modifying the PCM.

6. CONCLUSION

Within this paper, we presented a list of metamodel smells we found in the current version of the PCM. We identified 10 different types of smells. Simple metamodel errors (which cause validation errors and prohibit code generation) are not included, as the PCM is already in operation and thus does not contain any.

Two of the presented smells are exclusive to metamodels. The remaining eight smells also represent issues in OO. However, there are differences in the usage of metamodeling, OO design and code. Therefore, in OO there is not as much emphasis on the smells which were presented here.

The smells presented in this paper are specific to languages that are similar to EMOF. All smells are concerned with the following basic concepts: classes, relations and attributes. Some smells are concerned with more specific meta-language features: explicit containments (5.1, 5.2), modularization (5.3, 5.4, 5.5), inheritance (5.8, 5.9) and enums (5.10).

Four of the smells might be detected by scanning for antipatterns. However, all of them still have to be reviewed, as there are circumstances where an occurrence is not necessarily bad design. The remaining six smells can only be detected by manual review.

For each smell, we explain how it might come into being. Some smells are built in by mere carelessness or lack of knowledge. Therefore, knowledge of these metamodel smells is very valuable for metamodel developers. Other smells, however, do only manifest with time, when multiple evolution steps have been performed (some of them in a shortsighted manner).

For each smell, we explained the effects we observed and further consequences that we expected. Some smells add unnecessary complexity. Others simply impair understandability by obfuscating design decisions or the intended structure of the metamodel. Some have negative effects on coupling and cohesion of packages. Thus, these metamodel smells have detrimental effects on metamodel maintainability. The consequences are even worse, if the metamodel is long living, is evolved and the smells accumulate without being fixed. Metamodels tend to live in metamodel-centric software systems. Many tools, like editors, analyzers and simulators, are built upon them. If the metamodel is changed, all tools have to be fixed. The effort caused by resolving smells in the metamodel increases over time, as new dependencies pile up. Thus, smells should be fixed as early as possible.

Future work includes inspecting further metamodels also including less mature ones. Further, results from smell and error detection tool could be incorporated and analyzed.

7. ACKNOWLEDGMENTS

This work was supported by the Helmholtz Association of German Research Centers and the DFG (German Research Foundation) under the Priority Program SPP1593: Design For Future – Managed Software Evolution. We would like to thank Anne Koziolek, Richard Rhinelander, Kiana Rostami, Dominik Werle, Kateryna Yurchenko and the anonymous peer-reviewers for their valuable input.

References

[1] T. Arendt and G. Taentzer. “A tool environment for quality assurance based on the Eclipse Modeling Framework”. In: *Automated Software Engineering* 20.2 (2013), pp. 141–184.

[2] T. Arendt et al. “Defining and checking model smells: A quality assurance task for models based on the

eclipse modeling framework". In: *BENEVOL workshop*. 2010.

[3] V. Basili et al. "A validation of object-oriented design metrics as quality indicators". In: *Software Engineering, IEEE Transactions on* 22.10 (Oct. 1996).

[4] P. Bourque et al. *Guide to the software engineering body of knowledge*. IEEE, 2014.

[5] S. R. Chidamber and C. F. Kemerer. "Towards a Metrics Suite for Object Oriented Design". In: *SIGPLAN Not.* 26.11 (Nov. 1991), pp. 197–211.

[6] J. Di Rocco et al. "Mining Metrics for Understanding Metamodel Characteristics". In: *Workshop on Modeling in Software Engineering*. ACM, 2014.

[7] M. Elaasar. "An approach to design pattern and anti-pattern detection in mof-based modeling languages". PhD thesis. Carleton University Ottawa, 2012.

[8] M. Elaasar et al. "Domain-Specific Model Verification with QVT". In: *ECMFA*. Springer, 2011.

[9] M. Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] E. Freeman et al. *Head First Design Patterns*. Head First. O'Reilly Media, 2004.

[11] I. García-Magariño et al. "An evaluation framework for MAS modeling languages based on metamodel metrics". In: *Agent-Oriented Software Engineering* (2009).

[12] M. Genero et al. "Building measure-based prediction models for UML class diagram maintainability". English. In: *Empirical Software Engineering* 12 (5 2007).

[13] J. J. C. Gómez et al. "Searching the Boundaries of a Modeling Space to Test Metamodels". In: *Software Testing, Verification, and Validation, 2008 International Conference on* (2012), pp. 131–140.

[14] K. Julisch. "Understanding and overcoming cyber security anti-patterns". In: *Computer Networks* 57.10 (2013), pp. 2206–2211.

[15] R. Jung et al. "A Method for Aspect-oriented Meta-Model Evolution". In: *Proceedings of the 2Nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO '14. York, United Kingdom: ACM, July 2014, 19:19–19:22.

[16] R. Jung et al. "GECO: A Generator Composition Approach for Aspect-Oriented DSLs". In: *Theory and Practice of Model Transformations: 9th International Conference on Model Transformation, ICMT 2016*. Springer International Publishing, 2016, pp. 141–156.

[17] M. E. Kramer et al. "Extending the Palladio Component Model using Profiles and Stereotypes". In: *Palladio Days 2012*. Ed. by S. Becker et al. Karlsruhe Reports in Informatics ; 2012,21. Karlsruhe: KIT, Faculty of Informatics, 2012, pp. 7–15.

[18] P. Langer et al. "EMF Profiles: A Lightweight Extension Approach for EMF Models". In: *Journal of Object Technology* 11.1 (2012), 8:1–29.

[19] K. Lano and S. K. Rahimi. "Case study: Class diagram restructuring". In: *Proceedings Sixth Transformation Tool Contest, TTC 2013, Budapest, Hungary, 19-20 June, 2013.* 2013, pp. 8–15.

[20] J. J. López-Fernández et al. "Assessing the Quality of Meta-models". In: *Proceedings of the 11th Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa)*. 2014, p. 3.

[21] M. Manso et al. "No-redundant Metrics for UML Class Diagram Structural Complexity". In: *Advanced Information Systems Engineering*. Vol. 2681. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 127–142.

[22] M. Marchesi. "OOA metrics for the Unified Modeling Language". In: *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*. Mar. 1998, pp. 67–73.

[23] R. Martin. *Agile Software Development: Principles, Patterns, and Practices*. PH, 2003.

[24] Object Management Group (OMG). *MOF 2.4.2 Core Specification (formal/2014-04-03)*. 2014.

[25] Object Management Group (OMG). *Object Constraint Language, v2.0 (formal/06-05-01)*. 2006.

[26] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. to appear. Cambridge, MA: MIT Press, Oct. 2016. 408 pp.

[27] L. Rising. *The Patterns Handbook: Techniques, Strategies, and Applications*. SIGS, 1998.

[28] C. U. Smith and L. G. Williams. "Software performance antipatterns". In: *Workshop on Software and Performance*. 2000, pp. 127–136.

[29] D. Steinberg et al. *EMF: Eclipse Modeling Framework*. second revised. Eclipse series. Addison-Wesley Longman, Amsterdam, Dec. 2008.

[30] M. Strittmatter and R. Heinrich. "A Reference Structure for Metamodels of Quality-Aware Domain-Specific Languages". In: *13th Working IEEE/IFIP Conference on Software Architecture*. Apr. 2016, pp. 268–269.

[31] M. Strittmatter and R. Heinrich. "Challenges in the Evolution of Metamodels". In: *3rd Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems*. Vol. 36. Softwaretechnik-Trends 1. 2016, pp. 12–15.

[32] M. Strittmatter and M. Langhammer. "Identifying Semantically Cohesive Modules within the Palladio Meta-Model". In: *Symposium on Software Performance*. Universitätsbibliothek Stuttgart, Nov. 2014, pp. 160–176.

[33] M. Strittmatter et al. "A Modular Reference Structure for Component-based Architecture Description Languages". In: *Model-Driven Engineering for Component-Based Systems*. CEUR, 2015, pp. 36–41.

[34] R. Subramanyam and M. Krishnan. "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects". In: *IEEE Transactions on Software Engineering* 29.4 (2003).

[35] E. Vépa et al. "Measuring model repositories". In: *Proceedings of the 1st Workshop on Model Size Metrics*. 2006.

[36] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. 1st. Prentice-Hall, 1979.

Semantic-based Model Matching with EMFCompare

Lorenzo Addazi
Mälardalen University, IDT,
Västerås, Sweden
lai15004@student.mdh.se

Antonio Cicchetti
Mälardalen University, IDT,
Västerås, Sweden
antonio.cicchetti@mdh.se

Juri Di Rocco
University of L'Aquila
I-67100 L'Aquila, Italy
juri.dirocco@univaq.it

Davide Di Ruscio
University of L'Aquila
I-67100 L'Aquila, Italy
davide.diruscio@univaq.it

Ludovico Iovino
Gran Sasso Science Institute
I-67100 L'Aquila, Italy
ludovico.iovino@gssi.infn.it

Alfonso Pierantonio
University of L'Aquila
I-67100 L'Aquila, Italy
Mälardalen University, IDT,
Västerås, Sweden
alfonso.pierantonio@univaq.it,
alfonso.pierantonio@mdh.it

ABSTRACT

In MDE resolving pragmatic issues related to the management of models is key to success. Model comparison is one of the most challenging operations playing a central role in a wide range of modelling activities including model versioning, evolution and even collaborative and distributed specification of models. Over the last decade, several syntactic methods have been proposed to compare models even though they struggle in achieving higher levels of accuracy especially when the semantics of the application domain has to be considered. Existing methods improve comparison precision at the price of high performance costs.

This paper discusses a lightweight semantic comparison method, which relies on a new matching algorithm that considers ontological information encoded in the WordNet lexical database further than ordinary syntactical and structural correlations. The approach has been implemented as extension of EMFCompare and evaluated to measure its precision and performances when compared to existing approaches.

Keywords

model differencing; syntactic matching; semantic matching; ontological matching; EMFCompare

1. INTRODUCTION

Model-Driven Engineering (MDE) promotes the migration from a code-centric to a model-based approach to cope with the increasing development complexity of modern software systems. Models abstract real-world phenomena focusing on a specific aspect, e.g. its dynamic behaviour, its static structure, and assume the role of first-class artefacts throughout the software life cycle [1].

Addressing pragmatic issues related to the management and evolution of models has become a major concern in Model Driven Engineering [2] (MDE). Critical facilities such as model mergers and model difference tools related to model-level observability are key in distributed development of modeling artifacts. As noticed already in 2003 by Bran Selic model difference tools *"must work at a semantically meaningful level."* [3].

Over the years, relevant advances have been provided in the area of model differencing in terms of methods, such as similarity-based approaches [4, 5], and tools with the introduction of the EMFCompare [6] in the EMF ecosystem. However, the problem of determining model differences represents an intrinsically complex task, especially when dealing with two-way state-based comparisons [7], i.e. when the differencing is based on the sole information stored in the two models being compared.

Model differencing, indeed, relies on *Model Matching*, i.e. the calculation of correspondences among model elements, which can be reduced to the NP-Hard *Graph Isomorphism Problem*, that is the problem of finding correspondences between graphs [8, 5]. The available approaches to model matching are all different ways to deal with this intrinsic hardness, which is typically alleviated through either domain-specific or generic but approximate solution [9].

In this paper, we make a first step towards the introduction of semantic reasoning within the matching process in EMFCompare. In particular, we present a custom matching engine extending the default one of EMFCompare. In addition to the syntactical and structural correlations, the proposed extension compares model elements with respect to their semantic meaning using the WordNet lexical database [10]. Furthermore, the effectiveness and efficiency of the proposed approach is evaluated on a matching scenario based on an existing benchmark.

Structure of the paper: The paper is organized as follows. Section 2 presents an example motivating the work. Section 3 provides an overview of the model differencing problem. Section 4 outlines the main phases of the comparison process of EMFCompare and highlights its limitations. Section 5 outlines the WordNet lexical database, and the various semantic similarity measures based on its structural organization of concepts. Section 6 presents our main contribution, that is, a semantic extension of the matching process in EMFCompare. Section 7 evaluates the extension on a matching scenario based on an existing benchmark. Section 8 concludes the paper providing a brief summary and a discussion of possible future directions.

2. MOTIVATING SCENARIO

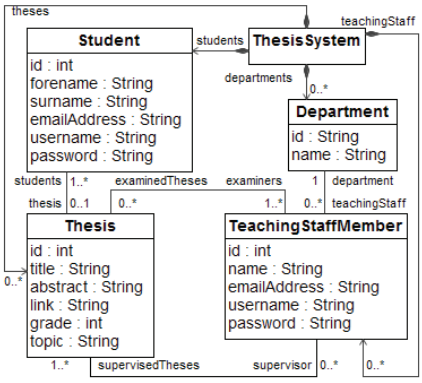
As mentioned before, the problem of calculating differences between two model versions is intrinsically difficult. Figure 1 depicts a practical example involving two different versions of the *ThesisManagementSystem* metamodel, that is a small-scale reproduction of university theses management portal. According to the initial version of the metamodel shown in Fig.1.a, a `ThesisSystem` contains information about `Department`s, `Student`s, teaching staff (`TeachingStaffMember`), and `Thesis`. In order to satisfy unforeseen requirements or to refine existing constructs, metamodels can be modified as in the case of the new version of the *ThesisManagementSystem* metamodel shown in Fig.1.b, which has

been obtained by applying the two metamodel changes described below.

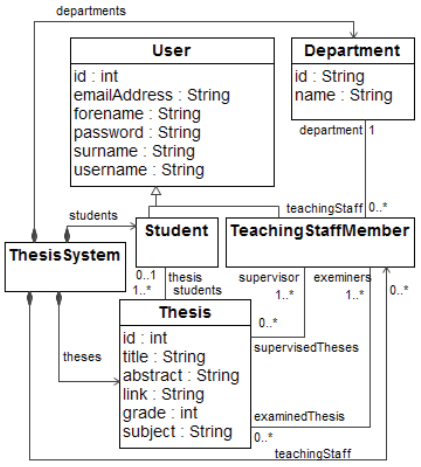
Extract user super class: in the first version of the metamodel, `Student` and `TeachingStaffMember` classes are completely separated despite sharing most of their attributes. Moreover, the metaclass `Student` has `forename` and `surname` attributes while `TeachingStaffMember` has only a `name` attribute grouping both name and surname. In light of this, the evolved metamodel version extracts the common information among `Student` and `TeachingStaffMember` into an additional superclass `User`, while `TeachingStaffMember`'s `name` is partitioned in `User`'s `forename` and `surname` attributes.

Thesis Attribute Renaming: in the first version of the metamodel, the main theme of a thesis is expressed through the `topic` attribute in the `Thesis` class. In the newer version, this attribute is renamed to `subject`. Intuitively, these attributes express the same concept, i.e. the principal issue discussed in a given thesis.

Table 1 reports the desired correspondences between the two metamodels, as determined by manually comparing the two versions. In particular, the entries named `S`, `T`, `D`, `Th`, `U` and `TS` correspond to `Student`, `TeachingStaffMember`, `Department`, `Thesis`, `User`, and `ThesisSystem` entities in the metamodels. For instance, there is a match between `Student` class in the initial and evolved metamodels; between the attributes `id`, `forename`, etc. in the initial metamodel, and the corresponding ones inherited by the class `User`, i.e. `U.id`, `U.forename`, etc. in the evolved version; between the relationships outgoing from `Student` and targeting the `Thesis` classes. The total number of manually identified correspondences is 37.



a) Initial metamodel version



b) Evolved metamodel version

Figure 1: A (meta-)model evolution scenario

Table 1: Manually identified matches

Element version 1	Element version 2
S	S
S.id	U.id
S.forename	U.forename
S.surname	U.surname
S.emailAddress	U.emailAddress
S.username	U.username
S.password	U.password
S.thesis	S.thesis
T	T
T.id	U.id
T.name	U.forename
T.name	U.surname
T.emailAddress	U.emailAddress
T.username	U.username
T.password	U.password
T.examinedTheses	T.examinedTheses
T.supervisedTheses	T.supervisedTheses
T.department	T.department
D[4 matches]*	D[4 matches]*
Th[9 mathches]•	Th[9 matches]•
Th.topic	Th.subject
TS[5 matches]*	TS[5 matches]*

* contained structural features are fully matched
• remaining structural features are fully matched

3. MODEL DIFFERENCING OVERVIEW

Existing model differencing approaches can be considered as logically composed of three main phases: *i)* compared models are imported in a differencing friendly format (typically graph-based structures); *ii)* a matching algorithm navigates the models to detect and establish correspondences between entities in the two models being compared; *iii)* a dedicated algorithm computes the differences of the matched elements in terms of (at least) additions, deletions, and changes of model entities as based on the matches established in the previous phase. In this respect, the matching phase is critical in any differencing approach, since any erroneous match, both false-positive and false-negative, results in a wrong output. More in general, requirements for model matching approaches include accuracy, a high level of abstraction at which the comparison is performed, independence from particular tools, domains, and languages, efficiency, and minimal effort. In [9] authors classify model matching approaches as follows:

– *Static Identity-Based Matching*: in this category, it is assumed that each model element has a persistent unique identifier that is assigned to it upon creation. Therefore, a basic approach for matching models is to identify matching model elements based on their corresponding identities (as in [11, 12, 13]);

– *Signature-Based Matching*: in this category, the identity of each model element is not static. Instead its identity, typically referred to as signature, is dynamically calculated by combining the values of its features. The signature computa-

tion is performed by means of a user-defined function specified using a model querying language (as in [14]);

– *Similarity-Based Matching*: the approaches in this category treat models as typed attribute graphs and attempts to identify matching elements based on the aggregated similarity of their features. It is worth noting that not all features of model elements are equally relevant for establishing a match (e.g. classes with matching names are more likely to be matched with classes specialising the same parent superclass). Therefore, similarity-based algorithms typically need to be provided with a configuration that specifies the relative weight of each feature and thus of detected correspondences (as in [5, 15, 13]);

– *Custom Language-Specific Matching Algorithms*: this category involves matching algorithms tailored to a particular modelling language. Notably, UMLDiff [4] and the work in [16] specifically target UML models and statecharts, respectively. In these cases, the narrow set of available entities, their semantics, and their valid evolution alternatives, allows to specialise the matching algorithm computations.

None of the approaches previously listed can be considered as the best solution as shown in [9]. Instead, the choice of a model matching solution should be evaluated as trying to optimise the trade-off between the constraints imposed by the context and the particular task at stake. This work is based on EMFCompare due to the flexibility and customisability of its matching engine, as detailed in the remainder of this section.

4. THE EMFCOMPARE TOOL: OVERVIEW AND LIMITATIONS

EMFCompare [13] is an Eclipse project which was initiated in 2006 at Eclipse Summit Europe, where the need for a model comparison engine emerged. It provides generic and customisable support for model comparison and merge of EMF models, such as Ecore and UML. The most relevant characteristic with respect to other approaches, at the time, consists in its high degree of extensibility: in fact, the whole comparison process is indeed designed to be completely customisable. In particular, the different activities composing the comparison process are explicitly disjoint and managed using different software entities, i.e. *engines*. Furthermore, according to its extensible nature, EMFCompare provides full support for the extension of its default metamodel-independent behaviour to tailor custom or metamodel-specific solutions.

The main advantage resulting from the explicit partition of the comparison process in its various sub-activities is that developers can focus on the specific part of the process they are going to extend/customise, while leaving the management of the remaining steps to the framework. For instance, the default matching approach adopted in EMFCompare falls in the *similarity-based* techniques. However, it also provides the possibility to adopt a *static identity-based* or a *signature-based* matching approach, possibly defining custom generator functions, if needed. In this regard, it is worth mentioning that this paper focuses on metamodel-independent differencing, which is incompatible with *static identity-based* and *signature-based* approaches [17].

In the following, we first provide a brief description of the overall comparison process in EMFCompare. Then, we focus our attention on the issues and limitations affecting the matching phase; such a phase will constitute the foothold on which our main contribution is built.

4.1 The Comparison Process

The EMFCompare comparison process can be roughly divided into six phases, whose main characteristics are singularly described below. The matching phase is illustrated in deeper details due to its relevance for this contribution, while the interested reader can refer to [13] for further details about the other phases.

Phase 1: Model resolving - This phase builds a logical representation of the overall comparison context, i.e. a model representing the artefacts to be compared enriched with the information required for the subsequent activities.

Phase 2: Model matching - Once resolved the logical models, the matching phase creates a set of *two-by-two* mappings while iterating over the model elements, e.g. `Student` in the model in Fig.1.a corresponds to *Student* in the model in Fig.1.b. By going into more details, for each element in the first model it is necessary to browse the elements in the second in order to find the most similar one. The default match engine firstly selects a specific element pair, and subsequently a similarity evaluation is computed by the combination of four different metrics in an overall score ranging from 0, i.e. completely different, to 1, i.e. identical elements. These metrics analyse the name of an element, its content, its type, and its relations with other elements.

Despite considering different characteristics of a given element, all the metrics adopt the same comparison approach. Indeed, for each metric, the final result is given by the execution of a string distance algorithm, e.g. the *Levenshtein Distance Algorithm* [18], on the string representation of the elements. In order to better explain this fundamental phase, listing 1 reports a simplified matching-algorithm pseudo-code. After the result initialisation (line 1), the algorithm iterates over all possible element pairs to compute their similarity (lines 2–4). It is worth noting that for optimisation purposes the models are not compared as a whole, whereas the possible elements are selected within a proper search window (line 3).

Once all the possible candidate matches are identified, the *createMatch* method takes as input the similarity values, and by using threshold policies produces the *Comparison* object containing all detected matches. The elements included in the current search window which are not matched yet are forwarded to future searches to possibly produce new matches.

Listing 1: EMFCompare default match engine implementation

```
1 result = Double[Model1.getElements().size()][Model1.
      getElements().size()]
2 foreach (elM1 : Model1.getElements())
3   foreach (elM2 : elM1.getWindowElements())
4     result[elM1][elM2] = calculateSimilarity(elM1, elM2)
5 return createMatches(result)
```

Phase 3: Model differencing - In this phase, the output of the matching is analysed to classify the various changes happened from a version to another. For instance, an element only present in the old version is classified as a deletion, an element only present in the new version as an addition, while a match can either be an untouched element, or an element subject to updates.

Phase 4: Detection of difference equivalences - During the equivalences step, the produced differences are analysed for filtering out redundant correspondences.

Phase 5: Detection of difference requirements - In this phase the produced differences are analysed once more to detect possible dependencies among them. Notably, the addition of the specialisation relationship between `Student` and `User` in the model in Fig.1.b could not exist without the creation of `User`, which does not exist in the old version of the model (see Fig.1.a).

Table 2: Matches identified by EMFCompare

Element version 1	Element version 2
S	S
S.surname	U.surname
S.thesis	S.thesis
T	T
T.examinedTheses	T.examinedTheses
T.supervisedTheses	T.supervisedTheses
T.department	T.department
D[4 matches]*	D[4 matches]*
Th [9 matches]•	Th [9 mathes]•
TS[5 matches]*	TS[5 matches]*

* contained structural features are fully matched
• `topic` and `subject` attributes are not paired but other contained structural features are fully matched.

Phase 6: Detection of difference conflicts - This phase allows to link the comparison process with a conflict detection mechanism, for instance when dealing with models managed through a Version Control System (VCS).

4.2 Limitations and Issues

According to performed experiments EMFCompare might have difficulties when dealing with comparison scenarios presenting complex correspondences. In particular, we define *complex* those differences which cannot be detected considering syntactical features only, or those differences characterised by n-to-m matches. Examples of representative complex correspondences are renaming an element using a synonym term, extracting features in common among multiple entities into a more generic one, or vice-versa, decomposing or distributing an entity feature among more specific entities.

By considering the default implementation of the EMFCompare matching process, we have classified the emerging issues into two categories, namely *contextual issues* and *linguistic issues*. The former result from the limited consideration of the features characterising the elements surrounding the compared ones, e.g. the classes containing a given pair of attributes. The latter, i.e. *linguistic issues*, result from the lack of a semantical evaluation of the features characterising the compared elements. Renaming a given class using a syntactically different name, for example, could lead to a false-negative, i.e. undetected correspondence. Analogously, a false positive, i.e. unexpected correspondence, could result when renaming a given class using a semantically different term, which however presents a strong syntactical similarity with another existing one.

Keeping in mind *contextual* and *linguistic* issues, Table 2 show the calculated matches with respect to the motivating example introduced in Sec.2. The EMFCompare default implementation identifies 25 matches, in particular 12 false negatives and 0 false positives. In particular, EMFCompare is not able to detect the correspondence between the attributes in the old `Student` metaclass and the new ones inherited from the extracted superclass `User`. Moreover, the comparison does not match the old `topic` attribute with the new `subject` attribute contained in `Thesis` metaclass.

5. WORDNET-BASED SIMILARITY

In this section, we provide an overview of the WordNet lexical database (see Sect. 5.1) and of related semantic similarity measures (see Sect. 5.2). Such techniques and tools underpin the improvement of EMFCompare proposed in Sect. 6 in order to mitigate the issues discussed in the previous section.

5.1 WordNet in a nutshell

WordNet [10] is a lexical database for the English language. It was created and is being maintained at the Cognitive Science Laboratory of Princeton University under the direction of psychology professor George A. Miller. In this database, English words are grouped into sets of synonyms called *synsets*, which also include a generic definition joining the contained words together and information about the semantic relationships connecting them to other synsets. The specific meaning of one word under a specific *Part-Of-Speech* (POS) is called a *sense*. Each synset has a *gloss* that defines the concept it represents. For example, the words *night*, *nighttime*, and *dark* constitute a single synset that has the following gloss: *"the time after sunset and before sunrise while it is dark outside"*. The purpose is twofold: to produce a combination of dictionary and thesaurus that is more intuitively usable, and to support automatic text analysis and artificial intelligence applications [19]. Synsets are connected to one another through explicit semantic relations. Some of these relations (hypernym, hyponym for nouns, and hypernym and troponym for verbs) constitute *is-a-kind-of* (holonymy) and *is-a-part-of* (meronymy for nouns) hierarchies. For example, tree is a kind of plant, tree is a hyponym of plant, and plant is a hypernym of tree. Analogously, trunk is a part of a tree, and we have trunk as a meronym of tree. While semantic relations apply to all members of a synset, because they share a meaning but are all mutually synonyms, words can also be connected to other words through lexical relations, including antonyms (i.e., opposites of each other) which are derivationally related, as well. WordNet provides also the polysemy count of a word, i.e. the number of synsets that contain the word. If a word participates in several synsets (i.e., has several senses) then typically some senses are much more common than others.

5.2 Semantic Similarity Measures

Semantic similarity measures might be used for performing tasks such as term disambiguation [20], as well as text segmentation, and for checking ontologies for consistency or coherence. All the currently available measures can be grouped into four classes [19]: i) *path-length* based, ii) *information-content* based, iii) *feature* based, and iv) *hybrid* measures. In the following, a brief explanation for each of these measure classes is provided with further emphasis on information-content based measures, which have been adopted to develop the proposed EMFCompare extension.

Path-length based measures. The main idea of path length based measures is that the similarity between two concepts is a function of the length of the path linking the concepts and the position of the concepts in the WordNet taxonomy. Although most path length based measures are simple to use, local density of pairs (i.e., frequency of the involved terms) fails to be reflected [21].

Feature based measures. Unlike the other measures, feature based measures are independent from the taxonomy and the subsumptions of the concepts. In particular, they attempt to exploit the properties of the ontology in order to obtain similarity values. Indeed, this kind of measures is based on the assumption that each concept is described by a set of words indicating its properties or features, such as glosses in WordNet. The more common characteristic two concepts have and the less non-common characteristics they have, the more similar the concepts are. However, it is worth noting that this kind of measures introduces a noteworthy computational delay into the overall process. Furthermore, feature based measures require a complete and correct feature set to work properly, which is not always an easy task to perform [21].

Information-content based measures. In information-content (IC) based measures, it is assumed that each concept includes a certain amount of information in WordNet. Similarity measures are based on such information content of each concept within the taxonomy. The more common information two concepts share, the more similar the concepts are. Lin's Measure [22] uses both the amount of information needed to state the commonality between the two concepts and the information needed to fully describe these terms. The similarity measure formula is defined as follows:

$$Sim_{Lin}(c_1, c_2) = \frac{2 * IC(lso(c_1, c_2))}{IC(c_1) + IC(c_2)} \quad (1)$$

where $IC(lso(c_1, c_2))$ is the information needed to state the commonality between c_1 and c_2, whereas $IC(c_i)$ is the information needed to represent the concept c_i.

In general, IC based measures attempt to exploit the information related to a given pair of concepts in order to evaluate their similarity. Therefore, how to obtain IC represents a crucial issue, which will directly affect the measure application performance. In [23], the IC value of a concept is a function of its number of hyponyms. The more hyponyms a concept has, the more abstract it is. That is to say, concepts with many hyponyms convey less information than concepts that are leaves in the taxonomy. A common issue encountered in adopting IC based measures consists in less precise evaluations as regards the structure information of concepts, which is fairly captured in path length based measures on the contrary. However, IC based measures provide a relevant improvement in terms of computational complexity, thus making it the convenient in our case.

Hybrid measures. The hybrid measures combine the above mentioned ones. In practice, many measures not only are able to combine the ideas presented above, but also the relations, such as is-a, part-of, and so on. For instance, the Rodriguez's measure [24] includes three parts: (i) synonyms set, (ii) neighbourhoods, and (iii) features. The similarity value of each part is assigned to a weight, and then summed together. Generally, both IC and path length based measures are integrated as parameters into hybrid functions, as it is has been proposed in [25].

6. EXTENSION OF EMFCOMPARE WITH SEMANTIC MODEL MATCHING

In this section, we propose an extension of EMFCompare aiming at addressing both the *linguistic* and *contextual* issues presented in Section 4.2. Furthermore, our solution introduces support for the definition of custom matching approaches using ontological descriptions in EMFCompare, instead of custom-tailored match engine implementations.

Thanks to the intrinsic extensibility of EMFCompare, our method has been developed focusing on the matching phase only, hence leaving untouched the previous and subsequent steps[1]. In particular, the matching process has been modified redefining the selection process of model elements, i.e. which elements to compare among each other, and the evaluation approach itself, i.e. the computation of the similarity value corresponding to a specific model elements pair. The proposed solution limits its semantic reasoning to the comparison among model element names, and relies on the WordNet lexical dictionary as ontological source. However, the knowledge is not directly retrieved from the WordNet database, but rather from an automatically generated graph. In this way, our method does not directly depend on the WordNet database structure itself, hence enabling the adoption of other ontologies.

In order to define our method, we have extended the concept of *Match* in the underlying EMFCompare comparison model by introducing the *Semantic Match* concept (see the `Semantic Match Engine` element in Fig. 2). The differences with respect to the ordinary *Match* consist in: (i) *semantic distance value*, i.e. each match carries information about the semantic distance among the encapsulated model elements, (ii) *container matches list*, i.e. each match carries references to the containers of the encapsulated model elements, and (iii) *content matches list*, i.e. each match carries references to the content of the encapsulated model elements.

The overall matching process, as illustrated in Listing 2, can be decomposed in three separated phases: (i) *exploration*, (ii) *evaluation* and (iii) *filtering*. In compliance with the default implementation in EMFCompare, the process starts receiving a logical representation of the compared models and terminates producing a set containing the expected matches.

```
1 function createMatches(Comparison comparison, List
      leftEObjects, List rightEObjects){
2  SemanticMatch root = createSemanticMatch(null, null);
3  exploreMatches(root, leftEObjects, rightEObjects);
4  evaluateMatches(root);
5  filterMatches(root, comparison);
6 }
```

Listing 2: Pseudo-code of the proposed semantic model match approach

The three steps underpinning the proposed semantic match are singularly described in the following.

Exploration: during the exploration phase, our method builds a labelled graph representation of the compared models. In such a representation, each node represents a *semantic match*, while each incoming or outgoing labelled edge represents a connection with its parents or children elements, respectively. The exploration process is listed in Listing 3. Given a starting pair of model elements, i.e. the `root` parameter, the exploration phase first iterates over their contained elements, (lines 2-3). For each pair of elements of the same type, our method creates a new `Semantic Match` node and connects it to the initial pair (lines 5-7). Finally, the same exploration process is recursively repeated on the created node (line 8).

```
1 function exploreMatches(SemanticMatch root, List
     leftEObjects, List rightEObject){
2  foreach(leftEObject : leftEObjects)
3   foreach(rightEObject : rightEObjects){
4    if(leftEObject.getClass().equals(rightEObject.getClass
        ())){
5     SemanticMatch current = createSemanticMatch(
         leftEObject, rightEObject);
6     current.addParent(root);
7     root.addChild(current);
8     exploreMatches(current, leftEObject.getContent(),
         rightEObject.getContent());
9    }
10  }
11 }
```

Listing 3: Pseudo-code of the Exploration phase

Evaluation: once the graph representation of the compared models is created, each `Semantic Match` node is integrated with the semantic distance value between its encapsulated elements, as illustrated in Listing 4. Once completed the evaluation of a given node (line 2), the same process is recursively repeated on its children elements (lines 4-6).

[1]The prototypical implementation of the EMFCompare semantic model matching extension is available at: https://github.com/MDEGroup/EMFCompare-Semantic-Extension .

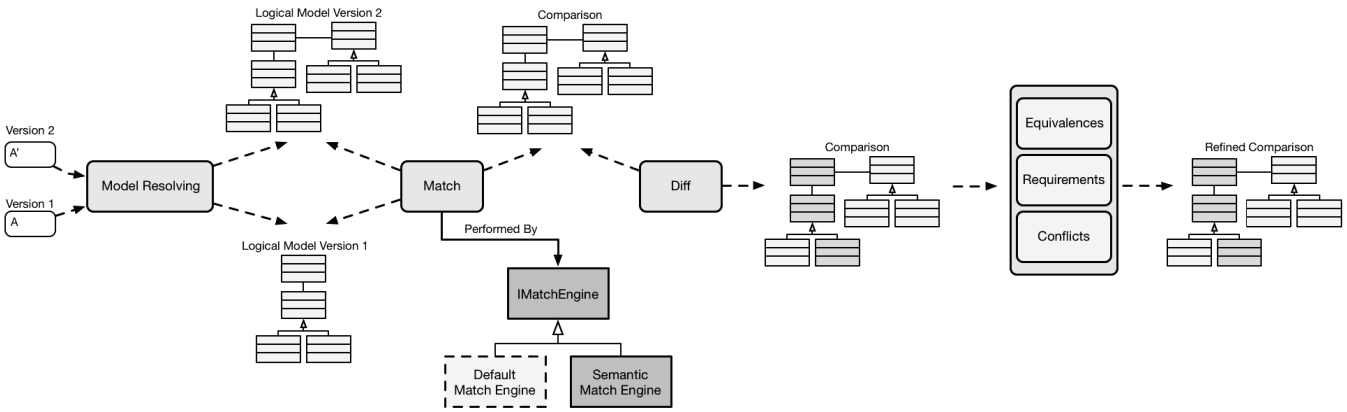


Figure 2: The proposed `Semantic Match Engine` in EMFCompare

```
1 function evaluateMatches(SemanticMatch root){
2  Double semanticDistance = evaluateSemanticDistance(root.
       getLeft(), root.getRight());
3  root.setSemanticDistance(semanticDistance);
4  foreach(child : root.getChildren()){
5   evaluateMatches(child);
6  }
7 }
```

Listing 4: Pseudo-code of the Evaluation phase

The semantic distance evaluation process is described in Listing 5. Given two model elements, the process begins with three subsequent preprocessing operations aimed to prepare the input for the subsequent evaluation (lines 2-3). Each string is tokenized by using as delimeters any non-alphabetical character, blank spaces and uppercasing - lowcasing changes in the word (line 12). Once, the obtained sequences are analyzed using the *Stanford Log-Linear Part-Of-Speech Tagger* [26], which assigns parts of speech, such as noun, verb or adjective, to each elements (line 13). Finally, in the token stemming step, for each tagged token, our method retrieves the corresponding root word using WordNet (line 14). At the end of the preprocessing phase, the initial strings are transformed to POS-tagged sequences, which represent the input for the actual comparison phase.

In order to evaluate the semantic similarity between two token sequences, our method first compares each element from the first with the elements from the latter (lines 5-6). For each element pair, the comparison consists in retrieving all the possible synsets which are related to the given words, evaluate their semantic distance using the inverted Lin's Algorithm (see Sect. 5.2), and returns back the minimum obtained value, i.e. how much the given words are similar in the best case (lines 7). In particular, we chose to adopt the corpora-independent method proposed in [23], which uses WordNet itself as a statistical resource to calculate IC values. Once that each pair has been evaluated, the final result is given by the *Min Match Average* of the obtained values, i.e. the sum of the minimum distance for each pair of tokens, divided by the maximum token list length among the involved ones (line 8). The final value ranges from 0, i.e. identical elements, to 1, i.e. not matching.

```
1 function Double evaluateSemanticDistance(EObject left,
      EObject right){
2  leftTokens = inputProcessing(left);
3  rightTokens = inputProcessing(right);
4  Double[][] result = new Double[leftTokens.size()][
       rightTokens.size()];
5  foreach(leftToken : leftTokens)
6   foreach(rightToken : rightTokens)
7    result[leftToken][rightToken] = invertedLin(leftToken,
         rightToken);
8  return minMatchAverage(result);
9 }
10
11 function List inputProcessing(EObject element){
12  List result = tokenizeString(element.getName());
13  result = tagTokenList(result);
14  return stemTokenList(result);
15 }
```

Listing 5: Pseudo-code of the semantic distance evaluation

Filtering: starting from the semantic distance values obtained during the previous steps, in this phase each `Semantic Match` node is analysed in order to decide whether or not to put it into the result set, as illustrated in Listing 6. Given an initial node, its analysis value results from the weighted arithmetic mean of the semantic distance value between the encapsulated model elements, the average semantic distance values between their children, and the minimum semantic distance value between their parents (lines 2-5). In order to be accepted, a `Semantic Match` node must have a lower analysis value with respect to a pre-defined threshold T_α. According to the requirements imposed by EMFCompare, the final matches are inserted into the *comparison* set (line 7). Furthermore, for each rejected node, we create two substituting matches having the left and the right element only, respectively (lines 9-10). The same process is then recursively repeated for each `Semantic Match` child (lines 12-14).

```
1 function filterMatches(SemanticMatch root, Comparison
      comparison){
2  Double rootDistance = root.getSemanticDistance() * root.
       getSemanticDistanceWeight();
3  Double childrenDistance = averageSemanticDistance(root.
       getChildren());
4  Double parentDistance = minSemanticDistance(root.
       getParents());
5  Double overallDistance = weightedArithmeticMean(
       semanticDistance, childrenDistance, parentDistance)
       ;
6  if(overallDistance < T_α){
7   comparison.add(root);
8  } else {
9   comparison.add(new SemanticMatch(root.getLeft(), null))
        ;
10  comparison.add(new SemanticMatch(null, root.getRight())
        );
11 }
12 foreach(child : root.getChildren()){
13  filterMatches(child, comparison);
14 }
15 }
```

```
16
17function Double weightedArithmeticMean(Double root,
      Double children, Double parent){
18  return W_R*root + W_C*children + W_P*parent;
19}
```

Listing 6: Pseudo-code of the Filtering phase pseudo-code

By considering the motivating example presented in Section 2, the proposed semantic approach identifies correctly the 37 manual correspondences and no *false positives* are found. Therefore, the calculated matches are identical to the manual correspondences and the first matches are exactly the same shown in Table 1.

7. VALIDATION

In order to validate our approach, we performed an experiment using a matching case benchmark inspired by [27] beyond successful tests on the motivating example presented in Section 2. This choice was due to the impossibility to reproduce our example on [27], and also to the opportunity of exploiting a larger experimental set. Once completed the experiment, we have compared our results to the principal state-of-the-art model matching tools. In detail, we have considered EMFCompare and *Atlas Model Weaver* (AMW), which represent the state-of-the-art matching tools in EMF. Furthermore, in order to extend our evaluation to matching tools in general, we have also included four existing approaches from the field of ontology and schema matching [28, 29, 30, 31], and the search-based approach proposed in [27]. Furthermore, we have compared our approach against *GAMMA* and [27] with respect to the difference computation performances.

In this section, we first describe the corpus of data used in our experiment, as well as the measures used for the evaluation. Then, we discuss the obtained results and compare them to the results produced using the other approaches. We refer the interested reader to [27] for further details about the results obtained by the other tools.

7.1 The Model Exchange Benchmark

The benchmark considered for our experiment consists of five structural modelling languages, namely UML 2.0, UML 1.4.2, Ecore, WebML and EER, as shown in Table 3. These metamodels present different characteristics with respect to both the size and used terminology. Indeed, as far as the metamodel size is concerned, the considered metamodels range from small-sized, e.g. EER, to large-sized, e.g. UML 2.0. Furthermore, while Ecore and UML metamodels use an object-oriented (OO) terminology, WebML and EER use database (DB) terminology.

All the possible pairs of the considered metamodels are considered as input to calculate the matches by means of GAMMA, EMFCompare, and the proposed approach named Semantic EMFCompare hereafter. In order to evaluate the quality of the produced match results, manual correspondences are reused from the previous studies using the INRIA alignment format provided in [27].

7.2 Measures

In order to evaluate the results produced by our approach, we consider them with respect to three different metrics from the information retrieval field: *Precision*, *Recall* and *F-Measure*. In this context, the *Precision* measure denotes the percentage of correctly matched elements with respect to all the proposed matchings as shown in equation 2.

$$Precision = \frac{|\{Retrieved\,Matches\} \cap \{Relevant\,Matches\}|}{|\{Retrieved\,Matches\}|} \quad (2)$$

Metamodel	Size	Terminology
UML 2.0 CD	158	OO
UML 1.4.2 CD	143	OO
Ecore	83	OO
WebML	53	DB
EER	23	DB

Table 3: Metamodels of the Model Exchange benchmark

The *Recall* measure, instead, indicates the percentage of correctly matched elements with respect to all the expected matchings. In other words, it measures how many correct matchings have been produced, as shown in equation 3.

$$Recall = \frac{|\{Retrieved\,Matches\} \cap \{Relevant\,Matches\}|}{|\{Relevent\,Matches\}|} \quad (3)$$

Finally, the *F-measure* combines both accuracy and recall in order to get an equally weighted average value of the measures. The corresponding formula is shown in equation 4.

$$F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (4)$$

It is worth noting that all the considered evaluation measures produce a numerical result, ranging from 0 to 1, where 0 corresponds to the worst and the 1 to the best possible value.

7.3 Results

In this section, we discuss the obtained results with respect to *Precision*, *Accuracy* and *F-Measure*, first, and time performance, then.

7.3.1 Quality Performance Results

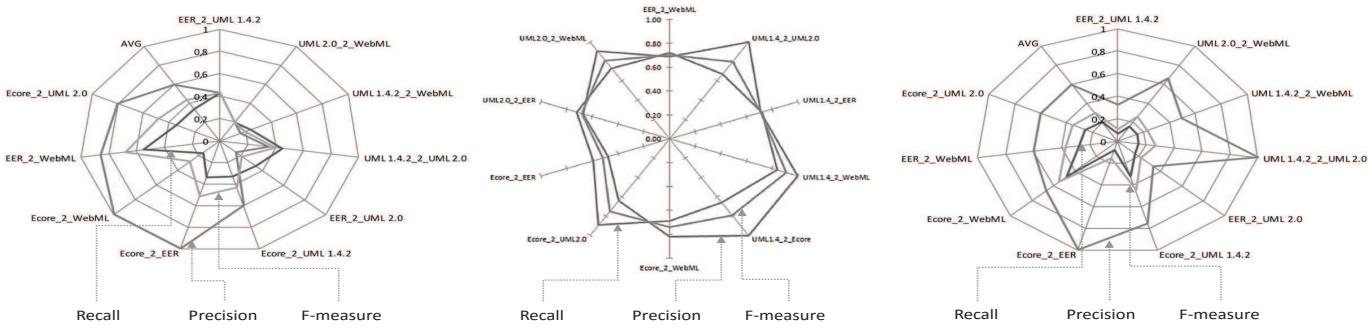
Figure 3 illustrates the results obtained with our proposed method, *GAMMA* and EMFCompare for the model exchange benchmark. Each axis of the glyph represents a matching task involving two metamodels from the initial set. The three metrics are represented using three quantitative variables, i.e. Precision, F-measure, and Recall.

Overall, our method provides the second best results with respect to *Precision*, *Recall* and *F-Measure*. The benchmark evaluation allows us to understand various characteristics about our solution, hence possible future improvements. First of all, the obtained results allow us to observe that our solution usually produces bigger number of matches than expected. In light of that using semantic approach, our extension matches more terms (i.e. *objects* and *item* by semantic extension and it does not by default implementation). Therefore *Precision* has in some cases a low value whereas considering *Recall* there is a significant improvement. However the *F-measure* has a better value than other tools analysed in [27].

Currently, the *GAMMA* [27] approach provides the best results in the state-of-the-art with respect to *Precision*, *Recall* and *F-Measure*. However, it is worth noting that *GAMMA* uses SBSE approaches to solve the matching problem, therefore the algorithm has to be initialised with a set of initial solutions which constitute the knowledge base for the computation.

7.3.2 Time Performance Results

Considering the main ideas driving the creation of EMFCompare, time performance is perhaps among the most important ones. Unlike for other approaches like [27], indeed, the tool has been de-



a) Semantic EMFCompare b) GAMMA c) EMFCompare

Figure 3: Comparison of the results related to the metamodels in the Model Exchange benchmark

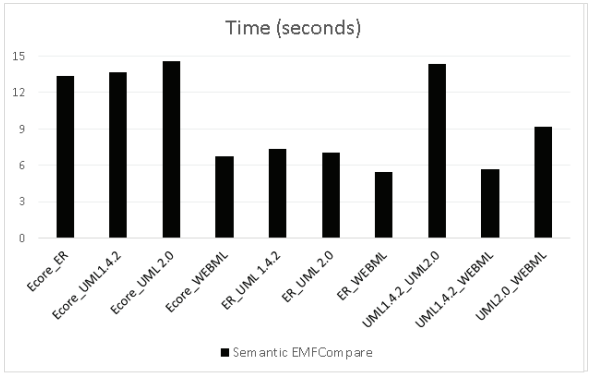


Figure 4: Execution time of Semantic EMFCompare

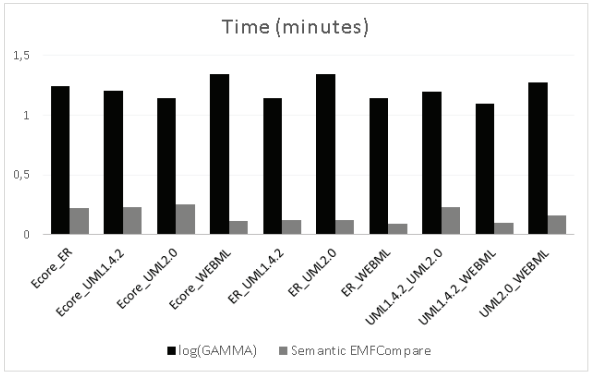


Figure 5: Comparison of execution time of GAMMA and Semantic EMFCompare

signed to provide a fast comparison algorithm, even at the price of showing some degradation with respect to the *Precision*, *Recall* and *F-Measure* of the produced results [6].

One of our main concerns was to introduce the semantic reasoning in the comparison process while keeping good time performances at the same time. As shown in figure 5, our extension successfully manages to perform a semantic reasoning while keeping acceptable time performances. In detail, our method works in terms of seconds as shown in 4, whereas for instance the solution proposed in [27] works in terms of minutes[2].

7.4 Discussion

The experience accumulated during the development of the proposed semantic extension of EMFCompare and its validation phase permits to draw some lessons learnt, which are detailed in this section.

Providing the comparison engine with a semantic reasoning can be done in a lightweight manner, as illustrated throughout this paper. However, we have noticed that an increasing matching power (thanks to the semantic reasoning) often comes to the price of an increasing imprecision, and in particular to a growing amount of false-positives or false-negatives that have to be traded-off. In fact, on the one hand the semantic reasoning is able to detect much more relationships between terms than a simple string comparison. On the other hand, the decomposition of entities' names not always produce terms known in the ontology exploited for the semantic reasoning (in our case WordNet), thus decreasing the similarity value. We tried to deal with this problem by tuning similarity thresholds and extending the information about the context in which the pair of elements is compared (i.e., including elements' owners and relationships). Nonetheless we feel that a more extensive experimentation is needed in order to achieve more reliable performances.

The issues mentioned above become very evident when comparing metamodels, since they are not expected to represent semantically related information, nor to contain concepts used in the usual English language. In this scenario, setting "looser thresholds" causes a lot of false-positives, while setting "tighter thresholds" remarkably reduces the number of matches. Therefore, quite surprisingly a pure syntactical comparison like the one computed by EMFCompare is able to provide on average more precise results than a differencing algorithm including some semantic reasoning. In this respect, even if comparing metamodels is a threat to validity, it is a factor that negatively affects the performances of the proposed extension. Therefore, we believe that our extension should be tested in the context of model comparison, given the higher probability of having semantically close names for the compared entities. Furthermore, we feel that the selection of the dictionary being used with respect to the kind of models to be matched plays a key role.

As a side remark, during our tests we have experimented the

[2]Please note that [27] comparison performances are rendered in a logarithmic scale for the sake of space.

lack of suitable mechanisms to generate differencing test cases. In fact, in order to validate the performances of our (and also other existing) differencing algorithm we would need an engine able to automatically produce different versions of a model and the corresponding expected comparison results.The current practice is to manually compare models and specify their differences, however this approach becomes quickly unfeasible with the increasing size of the input models. Alternatively, there exist tools and languages to generate sets of large models like Ecore Mutator[3] and Wodel [32] able to produce and track model manipulations (or mutations), however they typically work only on syntactical modifications. In this respect, there is the need to extend the mutation engine with the appropriate awareness of semantics variability, taking into account that there is always the risk of introducing some bias in the expected differencing results.

8. RELATED WORKS

Despite model versioning has been recognised as a critical feature for the successful adoption of MDE, performing model comparison with acceptable performances, both time- and precision-wise, is still an open research problem. In this respect, there exists a large body of literature discussing solutions to model comparison. A complete discussion of the existing literature goes far beyond the scope of this work, the interested reader can refer to [17, 27] as initial papers from which explore the field further. For the purpose of this paper, it is relevant to mention that in general approaches can deal with syntax or semantic matching. In both cases the comparison can be enriched by structural reasoning that helps in enhancing the precision of the matches [5]. Therefore, the extension proposed in this work can be considered as semantic matching with structural reasoning.

Other researchers have proposed semantic matching algorithms, like [33, 34, 35], just to mention a few. These techniques share the general approach of translating the syntax in a corresponding semantic domain, in which the matching is performed. Notably, both [34] and [35] rely on the behavioural semantics related to the compared models, while [33] proposes to translate the compared (meta-)models into corresponding ontologies and use ontology comparison algorithms. Our approach is closer to the last mentioned, since it exploits WordNet database and the linguistic relationships between terms to identify the possible semantic matches between the compared models. However, we do not consider the different methods as mutually exclusive, but rather potentially contributing to a more precise matching result. The open research problem is to provide empirical/formal foundations on how to appropriately combine the different techniques. As mentioned in this work, already combining syntactical and ontological matching is not straightforward in terms of the choice of the similarity thresholds, the order of the matching algorithms executions, and so forth.

The contribution by Kessentini et al. [27] is different from the previous ones since it is based on a search-based solution. Even if the precision of the matching can reach very good results, the unavoidable learning phase and the computation time can represent usability barriers of this and other similar solutions.

From a broader perspective, there exists a corpus of literature devoted to the problem of semantic clustering code to automatically extract trace links with requirements, documentation, design models, and more in general for information retrieval [36]. These techniques explore a set of artefacts, independently, with the goal of identifying clusters (also called topics), their characteristics (well-contained, cross-cutting, etc.), and the relationships among them. Then, the clusters in different documents are compared to identify matches [37]. Again, these approaches should not be conceived as mutually exclusive with respect to the exploitation of the semantic differencing illustrated in our proposal. On the contrary, they can constitute a useful characterisation of models for enhancing the precision of the matching algorithm.

[3]https://code.google.com/archive/a/eclipselabs.org/p/ecore-mutator

9. CONCLUSIONS AND FUTURE WORKS

In this paper, we presented an extension of the EMFCompare matching algorithm, which integrates the use of ontological information in order to calculate the similarity among two given model elements. The proposed solution represents an initial step towards the integration of semantic reasoning in the EMFCompare platform. Once presented our method, as well as the background notions it relies on, we also presented the results obtained from its application on the model exchange benchmark, which has been borrowed from [27]. Our solution does not provide the best results with respect to precision, recall and F-measure. However, the benchmark evaluation gives us useful information for possible future improvements. Moreover, we have successfully achieved to maintain fast time performance in our extension, therefore respecting one of the most important principles behind EMFCompare.

In the near future, we plan to further extend the semantic reasoning until completely cover the whole match engine. The current heuristics used in EMFCompare, indeed, are only reasonable when considering different versions of the same models, whereas tend to create inefficiencies whenever applied to models conforming to different metamodels. A possible solution to this inefficiency would be to allow the user to integrate an ontological specification of the differencing context, hence exploiting the consequent evaluations on its content.

Furthermore, although WordNet has been used in this paper, we have already defined our solution in order to be easily adaptable to different kind of ontological specifications. However, we plan to make our solution even more general in order to completely separate the ontological reasoning from its source description. Finally, it would be interesting to investigate the application of machine learning techniques in this context.

10. REFERENCES

[1] J. Bézivin, "On the unification power of models," *Software and Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[2] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006.

[3] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003.

[4] Z. Xing and E. Stroulia, "Umldiff: An algorithm for object-oriented design differencing," in *Procs of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2015)*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 54–65.

[5] Y. Lin, J. Gray, and F. Jouault, "DSMDiff: a differentiation tool for domain-specific models," *European Journal of Information Systems*, vol. 16, no. 4, pp. 349–361, 2007. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00483464

[6] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," *The European Journal for the Informatics Professional*, April-May 2008.

[7] T. Mens, "A state-of-the-art survey on software merging," *IEEE Trans. Softw. Eng.*, vol. 28, no. 5, pp. 449–462, May 2002.

[8] R. C. Read and D. G. Corneil, "The graph isomorphism disease," *J. Graph Theory*, vol. 1, no. 4, pp. 339–363, 1977.

[9] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Procs. of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, ser. CVSM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1 – 6.

[10] P. Oram, "Wordnet: An electronic lexical database. christiane fellbaum (ed.). cambridge, ma: Mit press, 1998. pp. 423." *Applied Psycholinguistics*, vol. 22, pp. 131–134, 3 2001.

[11] M. Alanen and I. Porres, *Difference and union of models*. Springer, 2003.

[12] P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel, "The topcased project: a toolkit in open source for critical aeronautic systems design," *Embedded Real Time Software (ERTS)*, vol. 781, pp. 54–59, 2006.

[13] A. Toulmé and I. Inc, "Presentation of emf compare utility," in *Eclipse Modeling Symposium*, 2006, pp. 1–8.

[14] R. Reddy, R. France, S. Ghosh, F. Fleurey, and B. Baudry, "Model composition-a signature-based approach," in *Aspect Oriented Modeling (AOM) Workshop*, 2005.

[15] C. Treude, S. Berlik, S. Wenzel, and U. Kelter, "Difference computation of large models," in *Procs. of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 295–304.

[16] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and merging of statecharts specifications," in *Procs. of the 29th Int. Conf. on Software Engineering (ICSE 2007)*. IEEE Computer Society, 2007, pp. 54–64.

[17] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Procs of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. IEEE Computer Society, 2009, pp. 1–6.

[18] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1091–1095, Jun. 2007.

[19] L. Meng, R. Huang, and J. Gu, "A review of semantic similarity measures in wordnet." *International Journal of Hybrid Information Technology*, vol. 6, no. 1, 2013.

[20] S. Patwardhan, S. Banerjee, and T. Pedersen, "Using measures of semantic relatedness for word sense disambiguation," in *Procs of the 4th Int. Conf. on Computational Linguistics and Intelligent Text Processing*, ser. CICLing'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 241–257.

[21] A. Budanitsky and G. Hirst, "Evaluating wordnet-based measures of lexical semantic relatedness," *Comput. Linguist.*, vol. 32, no. 1, pp. 13–47, Mar. 2006.

[22] D. Lin, "An information-theoretic definition of similarity," in *Procs of the 15th Int. Conf. on Machine Learning*, ser. ICML '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 296–304.

[23] N. Seco, T. Veale, and J. Hayes, "An intrinsic information content metric for semantic similarity in wordnet," 2004.

[24] M. A. Rodríguez and M. J. Egenhofer, "Determining semantic similarity among entity classes from different ontologies." *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 2, pp. 442–456, 2003.

[25] Z. Zhou, Y. Wang, and J. Gu, "New model of semantic similarity measuring in wordnet," in *3rd Int. Conf. on Intelligent System and Knowledge Engineering, (ISKE 2008)*, vol. 1, Nov 2008, pp. 256–261.

[26] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Procs of the 2003 Conf. of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, ser. NAACL '03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, pp. 173–180.

[27] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *J. Syst. Softw.*, vol. 97, no. C, pp. 1–14, Oct. 2014.

[28] J. Euzenat, "An api for ontology alignment," in *The Semantic Web – ISWC 2004*, ser. Lecture Notes in Computer Science, S. McIlraith, D. Plexousakis, and F. van Harmelen, Eds. Springer Berlin Heidelberg, 2004, vol. 3298, pp. 698–712.

[29] D. Aumueller, H.-H. Do, S. Massmann, and E. Rahm, "Schema and ontology matching with coma++," in *Procs of the 2005 ACM SIGMOD Int. Conf. on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 906–908.

[30] Y. Kalfoglou, B. Hu, D. Reynolds, and N. Shadbolt, "Capturing, representing and operationalising semantic integration (crosi) project - final report," University of Southampton, Technical Report, October 2005.

[31] M. Ehrig, "Foam - framework for ontology alignment and mapping; results of the ontology alignment initiative," in *Procs. of the Workshop on Integrating Ontologies. Volume 156., CEUR-WS.org (2005) 72–76*, 2005, pp. 72 – 76.

[32] P. Gómez-Abajo, E. Guerra, and J. de Lara, "Wodel: a domain-specific language for model mutation," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1968–1973.

[33] G. Kappel, H. Kargl, G. Kramler, A. Schauerhuber, M. Seidl, M. Strommer, and M. Wimmer, "Matching metamodels with semantic systems - an experience report," in *Datenbanksysteme in Business, Technologie und Web (BTW 2007), Workshop Proceedings, Aachen, Germany*, 2007, pp. 38–52.

[34] P. Langer, T. Mayerhofer, and G. Kappel, *Semantic Model Differencing Utilizing Behavioral Semantics Specifications*. Springer International Publishing, 2014, pp. 116–132.

[35] S. Maoz, J. O. Ringert, and B. Rumpe, "Summarizing semantic model differences," *CoRR*, vol. abs/1409.2307, 2014. [Online]. Available: http://arxiv.org/abs/1409.2307

[36] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230 – 243, 2007, 12th Working Conference on Reverse Engineering.

[37] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, Oct 2002.

Heterogeneous Megamodel Slicing for Model Evolution

Rick Salay
Department of Computer Science
University of Toronto
Toronto, Canada
rsalay@cs.toronto.edu

Sahar Kokaly
McMaster Centre for Software Certification
McMaster University
Hamilton, Canada
kokalys@mcmaster.ca

Marsha Chechik
Department of Computer Science
University of Toronto
Toronto, Canada
chechik@cs.toronto.edu

Tom Maibaum
McMaster Centre for Software Certification
McMaster University
Hamilton, Canada
maibaum@mcmaster.ca

ABSTRACT

Slicing is a widely used technique for supporting comprehension and assessing change impact during software evolution activities. While there has been substantial research into the slicing of particular model types, model-based software development typically involves heterogeneous collections of related models and there is little work addressing slicing in this context. In this paper, we propose a generic slicing approach for "megamodels" – a well-known model management technique for representing and manipulating collections of models and relationships between them. Our approach exploits existing model slicers for particular model types as well as the traceability relationships between models to address the broader heterogeneous model slicing problem. We illustrate our approach on an example of evolution in model-based automotive software development.

Keywords

Evolution, model slicing, model management, megamodels.

1. INTRODUCTION

Slicing is a widely used technique for supporting software evolution activities [19]. Specifically, *static* slicing [26] can identify the subset of software that is semantically dependent on a specific portion that has or is planned to be changed and hence is useful for assessing change impact due to evolution. In the MDE context, model slicing has been studied for particular model types, e.g., State Machines [16, 18], Class Diagrams [13, 18], etc. However, large-scale software systems are often described using heterogeneous collections of interrelated models, and change impact analysis requires a broader slicing approach that can address this.

While some work has addressed slicing for heterogeneous model collections, these have been limited to a specific set of model types (e.g., [20]) or remain at a theoretical level (e.g., [7]). In this paper, we propose a general and pragmatic static slicing algorithm for heterogeneous model collections. Specifically, (1) it operates on "megamodels" – a general modeling technique to represent collections of interrelated models; (2) it can work with arbitrary model types (e.g., conceptual, behavioural, goal models, test models, etc.) by utilizing their corresponding type-specific model slicers; and (3) it uses the widely adopted notion of traceability relations to assess change impact between models. We then analyze the proposed algorithm for termination, correctness, running time and minimality.

The remainder of this paper is structured as follows. In Sec. 2, we give a motivating example from the automotive software domain. In Sec. 3, we recall the background needed for the slicing approach and in Sec. 4, we describe the proposed slicing algorithm and its analysis. Then, in Sec. 5, we give a detailed illustration of the algorithm on the automotive example. In Sec. 6, we discuss related work and finally, in Sec. 7, we give our conclusions and report on future work.

2. MOTIVATING EXAMPLE

Consider an automotive subsystem that controls the behaviour of a power sliding door in a car. The system has an `Actuator` that is triggered on demand by a `Driver Switch`. This example is presented in Part 10 of the ISO 26262 standard [12]. Refer to Fig. 1 which shows the system models comprised of a Class Diagram (to model its structure), a Sequence Diagram (to model its behaviour) and a relationship between them. This can be visualized at a high-level as the megamodel (to be defined in Sec. 3) in Fig. 2.

The `Driver Switch` input is read by a dedicated Electronic Control Unit (ECU), referred to as `AC ECU`, which powers the `Actuator` through a dedicated power line. The vehicle equipped with the item is also fitted with an ECU which is able to provide the vehicle speed (VS). This ECU is referred to as `VS ECU`. The system includes a safety element, namely, a `Redundant Switch`. Including this element ensures a higher level of integrity for the overall system.

The `VS ECU` control unit provides the `AC ECU` with the vehicle speed. The `AC ECU` monitors the driver's requests, tests if the vehicle speed is less than or equal to 15 km/h, and if so, commands the `Actuator`. Thus, the sliding door can only be opened or closed if the vehicle speed is no more than 15 km/h. The `Redundant Switch` is located on the power line between the `AC ECU` and the `Actuator` as a secondary safety control. It switches on if the speed is less than or equal to 15 km/h, and off whenever the speed is greater than 15 km/h. It does this regardless of the state of the power line

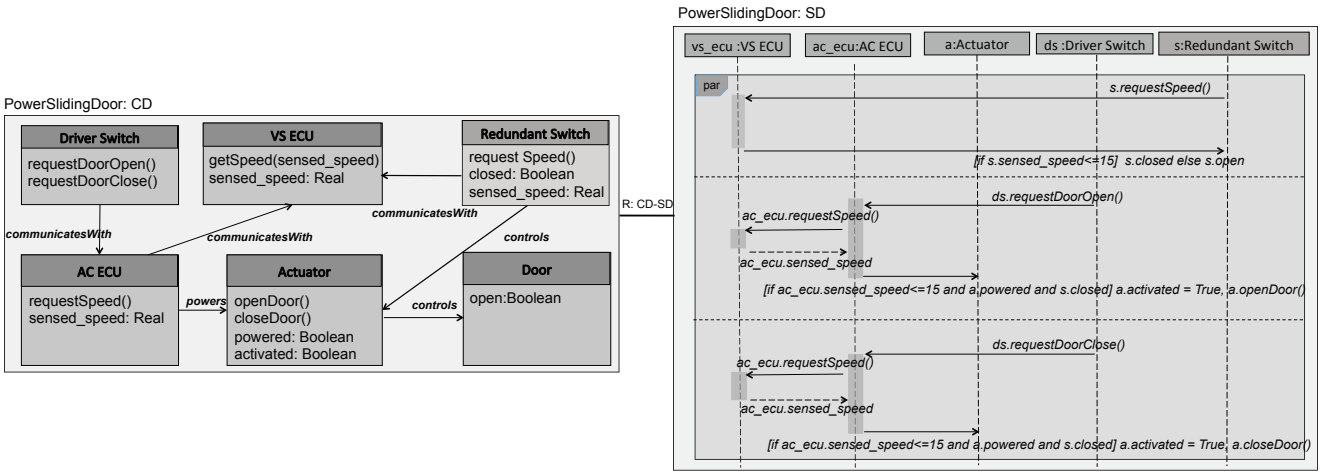


Figure 1: Power sliding door system models.

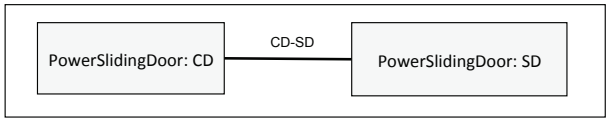


Figure 2: Power sliding door system megamodel.

(its power supply is independent). The `Actuator` operates only when it is powered.

Now, consider that the power sliding door system changes and the redundant switch is removed. This could be due to the need to minimize cost and produce a cheaper vehicle. In the new system, only the `AC ECU` checks the vehicle speed before commanding `Actuator`. In this case, it would be desirable to provide a sliced megamodel of the system that reflects the parts of the original megamodel affected by this change in order to help with system evolution activities. For example, we shown that with safety-critical software, such as for automotive systems, a system megamodel slice is an essential part of re-assessing the safety assurance of the system [15].

After presenting our slicing approach, we demonstrate it on the power sliding door example in Sec. 5.

3. BACKGROUND AND PRELIMINARIES

3.1 Megamodeling and Model Management

A complexity problem in MDE arises due to the proliferation of software models, and the area of Model Management [2] has emerged to address this challenge. Model management focuses on a high-level view in which entire models and their *relationships* (i.e., mappings between models) can be manipulated using *operators* (i.e., specialized model transformations) to achieve useful outcomes. In this paper, we focus on one of the model management operators – model *slice* [20]. Other model management operators that have been studied include *match* [2], *diff* [2], *lift* [23], and *merge* [6]. Each of these model management operators can be viewed as an *abstract* transformation that defines a class of concrete transformations, i.e., the implementations that refine the operator for particular model types. For example, a slice operator for class diagrams is implemented differently than a slice operator for state machines.

Megamodels. To help visualize and work with collections of models and their relationships, model management uses a special type of model called a *megamodel* [3]. In this paper, we define this and related concepts as follows.

Definition 1 (Megamodel). *A* megamodel *is a model with elements representing models and links between elements representing relationships between the models.*

Fig 3 shows the simplified metamodel used for megamodels in this paper. A `Megamodel` consists of a graph of named and typed `Model` elements with `Relationship` links connecting them. These refer to models and model relationships (defined below), respectively, and the types indicate their metamodels. The well-formedness constraint requires that the models on either end of every relationship are distinct. We have made the simplifying assumptions that (1) all relationships are binary; and (2) megamodels cannot be nested or reference other megamodels. In Sec. 4.3, we discuss how are these assumptions can be relaxed.

Fig. 2 shows a megamodel for our power sliding door example. Here, `PowerSlidingDoor : CD` and `PowerSliding−Door : SM` refer to the Class Diagram and Sequence Diagram, respectively, while the line connecting them refers to the relationship of type `CD − SD` for connecting these two types of models. Note that relationship names are optional.

Definition 2 (Model). *A* model *is set of typed elements and links conforming to a metamodel. We use the term* atom *to denote either an element or a link.*

Definition 3 (Model relationship). *A* model relationship *connecting models M and M' consists of a set of typed links conforming to a metamodel. Each link connects atoms of M to atoms of M'.*

Definition 4 (Traceability relationship). *A* traceability relationship *is a model relationship in which the links express a* dependency *relationship between the atoms it connects. The dependency relationship can be unidirectional or bidirectional depending on the type of traceability link.*

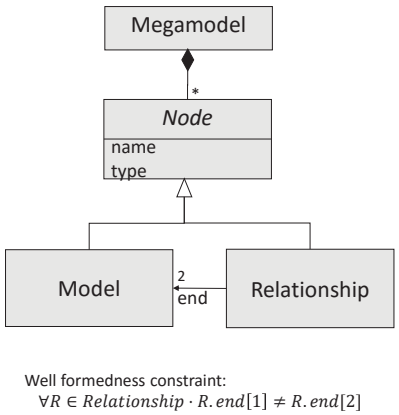


Figure 3: (Simplified) metamodel for megamodels.

Note that the definition of traceability relationship used in this paper is broader than that typically used by requirements engineering [11] and narrower than what is sometimes used for general modeling [1]. We focus solely on traceability relationships and use them to determine cross-model dependencies. In fact, we assume that the only relationships in the megamodels are the traceability relationships. In Sec. 4.3, we discuss how this assumption can be relaxed.

Definition 5 (Model fragment). *A* model fragment *S of a model M, denoted $S[M]$, is any subset of atoms of M.*

Definition 6 (Megamodel fragment). *A* megamodel fragment *S of a megamodel X, denoted $S[X]$, is any set of model fragments of the models in X. We say that $S[X]$ is contained in $S'[X]$, denoted $S[X] \sqsubseteq S'[X]$, iff the following condition holds:*

$$\forall M \in X \cdot \cup\{S[M] | S[M] \in S[X]\} \subseteq \cup\{S'[M] | S'[M] \in S'[X]\}$$

Thus, $S[X] \sqsubseteq S'[X]$ when, for each model $M \in X$, the combined model fragments of M in $S[X]$ is contained in the combined model fragments of M in $S'[X]$. Note that a megamodel fragment is defined as containing only model fragments and no relationships between them.

3.2 Model slicing

Program slicing, and, correspondingly, model slicing approaches, fall into four categories: *static*, *dynamic*, *conditional* and *amorphous* [7]. In each case, we are given a model and a slicing criterion indicating some "aspect of interest" in the model, and the slicing process produces a slice of the model that addresses the criterion. Static slicing uses a model fragment as the criterion. A *forward slice* expands the criterion to all dependent atoms while a *backward slice* expands to all depending atoms. While static slicing uses a subset of the syntax as a criterion, dynamic slicing uses a constraint from the semantic domain. For example, dynamic slicing can be used to identify the classes used in a particular run of a program. Conditional slicing combines both static and dynamic approaches by allowing a hybrid criterion. Finally, while the first three types of slicing produce a slice that is a fragment of the model, amorphous slicing allows the slice to be a different model. For example, the approach used in [20] adds stuttering transitions to state machine slices in order to preserve behaviours.

In this paper, we focus on static forward slicing since it is readily applicable to assessing the impact of changes due to model evolution. We define this as follows.

Definition 7 (Static forward model slice). *Given a model M and model fragment $S[M]$, the* static forward slice *of M with respect to the slicing criterion $S[M]$ is the model fragment $S'[M]$ satisfying the following conditions:*

1. *(Correctness) $S'[M]$ contains all atoms of M that are directly or indirectly dependent on atoms of $S[M]$.*
2. *(Minimality) Every atom of $S'[M]$ is either directly or indirectly dependent on atoms of $S[M]$.*

Note that since $S[M]$ is dependent on itself, these conditions imply that $S[M] \subseteq S'[M]$.

4. MEGAMODEL SLICING

In this section, we propose a slicing approach for heterogeneous megamodels. Intuitively, such a slicer should allow the criterion to be expressed as a megamodel fragment and the forward slice should expand this to the megamodel fragment containing all dependent elements. We generalize Def. 7 to capture this intuition.

Definition 8 (Static forward megamodel slice). *Given a megamodel X and megamodel fragment $S[X]$, the* static forward slice *of X with respect to slicing criterion $S[X]$ is the megamodel fragment $S'[X]$ satisfying the following conditions for all $M \in X$:*

1. *(Correctness) There exists a model fragment $S'[M] \in S'[X]$ that contains all atoms of M that are directly or indirectly dependent on the atoms of any model fragment in $S[X]$.*
2. *(Minimality) If there exists a model fragment $S'[M] \in S'[X]$, then every atom of $S'[M]$ is either directly or indirectly dependent on the atoms of some model fragment in $S[X]$.*

There are two levels of expansion in this slicing process: (1) expansion within individual models to dependent elements and, (2) expansion between models across relationships to dependent elements in neighbouring models. This two-level process is repeated until it produces no further expansion. For (1), we leverage existing type-specific slicers that take the semantics of the individual model types into account. For (2), we use the links in traceability relationships to connect dependent elements. Here, no special relationship-type specific slicers are needed since all relationship types are assumed to be sets of links and every link is assumed to represent a dependency.

Note that this definition of slicing is a *deep* slicing since the process includes the content of the models and relationships referenced by the elements of the megamodel. In contrast, a *shallow* megamodel slicing would be one that only considered the elements of the megamodel and not what they reference. Here, a subset of a megamodel (the criterion) is expanded to the subset that is connected directly or indirectly via relationship links (the slice), i.e., the shallow slice is the largest subset contained in the transitive closure of the initial subset taken along relationship links. There may be some use cases in which shallow megamodel slicing is useful but in this paper we focus on the deep version.

4.1 Slicing algorithm

Fig. 4 gives the algorithm for forward slice. The input is megamodel X with megamodel fragment $S_c[X]$ given as the slicing criterion. The output is megamodel fragment $S[X]$ representing the forward slice. The algorithm makes the following assumptions:

ASSUMPTION 1. *For each model type T represented in X, we have a slicer* $\texttt{Slice}_T$ *for models of type T that satisfies Def. 7.*

ASSUMPTION 2. *The set of traceability relationships in X express all and only the direct dependencies between atoms of models in X.*

In addition, we require several simple supporting operations.

DEFINITION 9 (`Union`). *Given a pair of megamodel fragments $S_1[X], S_2[X]$, the* megamodel fragment union*, denoted* $\texttt{Union}(S_1[X], S_2[X])$*, is defined with the following condition.*

$$\forall S[M] \in \texttt{Union}(S_1[X], S_2[X]) \cdot$$
$$S[M] = \cup\{S'[M] | S'[M] \in S_1[X] \cup S_2[X]\}$$

Thus, the $\texttt{Union}(S_1[X], S_2[X])$ can be constructed by first taking the set union $S_1[X] \cup S_2[X]$ and then unioning all model fragments of the same model within this.

DEFINITION 10 (`Trace`). *Given a traceability relationship R with ends M and M', and model fragment $S[M]$, the* trace *of $S[M]$ along R, denoted* $\texttt{Trace}(R, S[M])$ *is the model fragment $S'[M']$ consisting of the subset of atoms in M' dependent on the atoms in M according to R.*

We compute $\texttt{Trace}(R, S[M])$ by following the links of R from the atoms of M to the atoms of M'.

DEFINITION 11 (`OppEnd`). *Given a traceability relationship R with ends M and M', we define* $\texttt{OppEnd}(R, M) = M'$ *and* $\texttt{OppEnd}(R, M') = M$.

In line 1 of the algorithm, the current slice is initialized to the criterion. The two levels of expansion are in lines 4-9 and lines 10-17, respectively, inside the main loop of lines 2-19. For level 1, the temporary result $S_1[X]$ is initialized in line 3 to the empty set and then lines 5-9 iterate through the model fragments in the current slice. In line 7, the model type-specific slice is computed using the model fragment as the criterion and the result is accumulated in $S_1[X]$ (line 8).

The level 2 expansion temporary result $S_2[X]$ initialized on line 10. The outer iteration (lines 11-17) is over the model fragments from the level 1 expansion, and the inner iteration (lines 12-16) is over each relationship connected to the model fragment. Note that the set of relationships connected to a model fragment $S_1[M]$ is the set of relationships connected to M via the `end` property (see Fig. 3). For each such relationship R, we first determine the model M' on the other end of the relationship using supporting function `OppEnd` in line 13. Then in line 14, the model fragment $S_2[M']$ is produced by tracing the links in R from $S_1[M]$ to M'. Finally, in line 15, this result is accumulated in $S_2[X]$.

Algorithm: Forward Megamodel Slice
Input: megamodel X, criterion megamodel fragment $S_c[X]$
Output: slice megamodel fragment $S[X]$

```
1:  S[X] := S_c[X]
2:  do {
3:      S'[X] := S[X]
4:      S_1[X] := ∅
5:      for (S[M] ∈ S[X]) {
6:          T := M.type
7:          S_1[M] := Slice_T(M, S[M])
8:          S_1[X] := Union(S_1[X], {S_1[M]})
9:      }
10:     S_2[X] := ∅
11:     for (S_1[M] ∈ S_1[X]) {
12:         for (R ∈ M.end) {
13:             M' := OppEnd(R, M)
14:             S_2[M'] := Trace(R, S_1[M])
15:             S_2[X] := Union(S_2[X], {S_2[M']})
16:         }
17:     }
18:     S[X] := Union(S_1[X], S_2[X])
19: } until (S[X] ⊑ S'[X])
20: return S[X]
```

Figure 4: Algorithm for forward megamodel slice.

After the two levels of expansion, the combined result is computed in line 18 and checked to see if any actual expansion has occurred (line 19). If no expansion has occurred, a fixed point has been reached and the main loop exits with the current slice returned as the final result in line 20; otherwise, the main loop repeats.

4.2 Analysis

We consider the issues of termination, complexity and correctness for forward slice algorithm in Fig. 4.

Termination. We show that the slicing algorithm is guaranteed to terminate. After the level 1 expansion loop completes (lines 5-9), it is clear that $S'[X] \sqsubseteq S_1[X]$ since $S_1[X]$ is constructed by expanding each model fragment in the current slice $S[X]$ using type-specific slicers (see Assumption 1) and doing `Union` (see Def. 9). Furthermore, $S'[X] = S[X]$ (line 3). Then, in line 18, when the new slice is computed, $S_1[X] \sqsubseteq S[X]$ since `Union` cannot produce a result smaller than its arguments. Therefore, $S'[X] \sqsubseteq S[X]$. Thus, on line 19, either no expansion has occurred ($S[X] \sqsubseteq S'[X]$) and the algorithm terminates or some expansion has occurred and the loop iterates again. Thus, in each iteration, the current slice can only get larger and since this process is bounded by X, the algorithm must terminate.

Time Complexity. The level 1 loop (lines 5-9) can iterate N_M times and the level 2 loop (lines 11-17) can iterate N_M^2 times where N_M is the number of models in X. The dominating operation in the level 1 loop is the type-specific slicer. Since the time complexity varies according to the slicer used, we represent it using a type-independent upper bound $SL(n)$ as a function of the number of elements n in the input model. Tracing along a relationship and union (lines 14-15) is $O(N_a)$ in the worst case, where N_a is the total number of atoms across all models of X. Thus, in the worst case, one iteration of the main loop is $O(N_M \times SL(N_a) + N_M^2 \times N_a)$.

Finally, in the worst case, the size of the current slice can increase by one in each iteration of the main loop, for N_a iterations. Thus, the time complexity is given by:

$$O(N_a \times N_M \times SL(N_a) + N_M^2 \times N_a^2)$$

Correctness. We argue that the slicing algorithm satisfies the correctness condition in Def. 8. Assume that the algorithm is at line 3 and there exists a non-empty set of atoms not in the current slice $S[X]$ that are dependent on atoms of model fragments in $S[X]$. Note that if some atom a is indirectly dependent on an atom a', then there must be a sequence of directly dependent atoms $a, a_1, ..., a_n, a'$ connecting them. Thus, there must also be a non-empty set of atoms not in $S[X]$ that are *directly* dependent on atoms of model fragments in $S[X]$. Let us choose one such atom a' in some model M' in X that is directly dependent on an atom a in some model fragment $S[M]$ in $S[X]$. We consider the two cases: $M' = M$ and $M' \neq M$.

Case 1). If $M' = M$, then by Assumption 1, the slicer used in line 7 satisfies the correctness condition in Def. 7 and thus, atom a' will be added to a model fragment in $S_1[X]$ in an iteration of the level 1 loop (lines 5-9).

Case 2). If $M' \neq M$, then by Assumption 2, there is a traceability relationship R in X with a link that connects a to a' and thus, line 14 will cause a' to be added to a model fragment in $S_2[X]$ in an iteration of the level 2 loop (lines 11-17).

In either case, the atom a' will enter the next iteration of the slice in line 18. Furthermore, since the addition of a' expands the slice, the main loop will iterate again and will capture the next set of directly dependent atoms, and so on. When the set of directly dependent atoms not in $S[X]$ is empty, no further level 1 or level 2 expansion is possible, and the algorithm terminates.

Minimality. We show that the slicing algorithm satisfies the minimality condition in Def. 8. To do this, we must show that the slice produced by the algorithm contains no atom that is not dependent on the criterion. Assume that there is an atom a' in the final slice that is not dependent on the criterion. In this case, a' must have been added to the slice on line 7 or line 14 in some iteration of the main loop. However, by Assumption 1 and Def. 7, $\texttt{Slice}_T$ can only produce minimal model slices in line 7 and so a' could not have been added there. Also, by Assumption 2, traceability relationships only contain links between true dependencies and in line 14, `Trace` is applied from the current slice to these dependent atoms. Thus, a' could not have been added at line 14. Therefore, we have a contradiction and so the megamodel slice must be minimal.

4.3 Discussion

Well-formedness and referential integrity. Def. 7 does not require that a slice be a well-formed model. However, in practice, ensuring that a slice is well-formed may be desirable because the slice can be used directly by tools such as editors, analyzers and transformations. Making a model fragment into a well-formed model requires it to be expanded by a minimum number of atoms in order to satisfy the well-formedness constraints. For example, if a CD fragment contains an association without one of its endpoints, adding the missing endpoint class will make it well-formed.

The problem with doing this expansion is that atoms can be added that are *not dependent on the criterion* since, if they were dependent, then they would already be in the slice. In particular, if $\texttt{Slice}_T$ used in line 7 of the slicing algorithm always included an expansion to well-formedness then in the subsequent steps of the algorithm the atoms added for well-formedness would be treated as though they were atoms added for dependency. This would result in a non-minimal megamodel slice. As a result, we view the expansion to well-formedness as an *optional post-processing step* that could be applied after the megamodel slice is computed.

A similar argument can be made about the issue of *referential integrity*. Assume that one atom references another, e.g., a lifeline in a sequence diagram references the class of the object that the lifeline represents. The referenced class is not dependent on the referencing lifeline; thus, if the forward slice includes the lifeline, it need not contain the class. However, it may be desirable to expand the slice to include the class to provide relevant contextual information for the lifeline. As with well-formedness, this referential integrity expansion can introduce atoms that are not dependent on the criterion and thus such an expansion should only be done as a post-processing step on the slice.

Generalizing the slicing algorithm. In Sec. 3, we made several simplifying assumptions in order to focus on the core aspects of the slicing algorithm. We now briefly discuss how to relax these assumptions.

• **N-ary Relationships.** We have assumed that all relationships in the megamodel are binary but it is straightforward to extend the algorithm to handle N-ary relationships. Specifically, the iteration through the relationships (lines 12-16) must be generalized to handle the case where a traceability link holds between atoms in models on multiple `end`s, and the supporting operations `OppEnd` and `Trace` must be adapted to address this.

• **Nested megamodels.** In the general case, a megamodel can contain other megamodels. Such a megamodel could be viewed as a tree with models as leaves and nested megamodels as intermediate nodes. A megamodel fragment is a tree with the same structure but with model fragments as leaves. Thus, the algorithm follows a similar approach as currently but in addition it must preserve the megamodel tree structure in the final slice.

• **Arbitrary relationships.** We have assumed that all relationships are traceability relationships since these are the only ones that matter to the slicing algorithm. In general, however, there may be other types of relationships in the megamodel, e.g., refinement, overlap, etc. The simplest way to allow these relationship types is to ignore all non-traceability relationships in the loop in lines 12-16.

5. POWER SLIDING DOOR EXAMPLE

In this section, we demonstrate our slicing approach on the power sliding door example presented in Sec. 2.

5.1 Megamodels of class and sequence diagrams

For the purpose of the example presented here, we instantiate our general framework such that its input is a system megamodel X given by a class diagram `CD`, a sequence diagram `SD`, and a relationship $\texttt{CD} - \texttt{SD}$ between them. Note that, although these are both UML diagrams, we are treat-

Table 1: Dependency relations for CD and SD slicers.

Rule	Component under assessment	Dependant parts potentially impacted
CD1	Class	Owned attributes and methods. Associations connected to class. Attributes/methods in other classes using types introduced in this class. Subclasses.
SD1	Term (portion of an expression)	Associated expression.
SD2	Expression (guard/action)	Associated message.
SD3	Message	Associated arrow (from source to target lifeline).
SD4	Arrow	Arrows directly after the arrow in the sequence. Message on the arrow.
SD5	Lifeline	Arrows connected to the lifeline. Messages on arrows connected to the lifeline.

ing them separately for the sake of this example. In general, not all models in a megamodel have to be UML diagrams.

Assume we are given some known change on the megamodel, which represents the slicing criterion $S_c[X]$ used as input to our algorithm. As stated in Sec. 4, we also assume that we are provided with correct class diagram and sequence diagram model slicers similar to those presented in [18] and [21], respectively.

For simplicity, we define our own CD and SD slicers for this example as follows:

• CD slicer works with the dependency rule shown as CD1 in Table 1: If a class is being considered for impact assessment, then all of its attributes, methods, associations linked to it and its subclasses are considered dependant on it and could potentially be impacted. They are therefore to be added in the slice.

• SD slicer works with the dependency rules shown as SD1 − SD5 in Table 1: If a term, i.e., any portion of an expression (e.g., a guard or an action) in a message, is being considered for impact assessment, then its associated expression could be impacted. Similarly, if an expression (e.g., a guard or an action) is being considered, its associated message should be included in the slice. Other rules for impact assessment of messages, arrows and lifelines are shown in the table.

Note that both slicers satisfy Def. 7, i.e., they are correct and minimal. We also assume that the set of traceability relationships in CD − SD expresses all and only the dependencies between the CD and SD in our system megamodel.

5.2 Slicing of Power Sliding Door megamodel

Recall the Power Sliding Door megamodel presented in Fig. 2 which we refer to as PSD. The models represented by PSD are in Fig. 1.

There are three threads running in parallel in the sequence diagram: the top thread describes the behaviour of the Redundant Switch; the middle thread describes the behaviour when the driver requests to open the door, and the bottom thread describes the behaviour when the driver requests to close the door. The relationship R : CD − SD is a unidirectional traceability relationship (refer to Sec. 3) that goes from SD to CD, since the objects and terms of SD are dependent on classes, attributes and methods in CD. The traceability between the two models is given implicitly by the SD referencing parts of the CD.

As described in Sec. 2, let us consider a scenario where the system changes, and the redundancy is removed by deleting the Redundant Switch class from the CD. This change represents our slicing criterion given by the megamodel fragment with detail shown in Fig. 5. Note that only the class itself is considered for the impact assessment and not its methods, attributes and associations linked to it.

We now demonstrate the application of the forward megamodel slice algorithm presented in Fig. 4 on the megamodel PSD and the criterion megamodel fragment S_c[PSD].

Line 1 (Initialization): The current slice is initialized to the criterion S_c[PSD] shown as the highlighted parts of Fig. 5.

1st iteration of the outer loop (lines 2-19):

Lines 4-9 (Expansion Level 1): The temporary result S_1[PSD] is initialized to the empty set. Then in lines 5-9, we iterate through the model fragments in the current slice shown in Fig. 5. The CD is considered first and the CD slicer is used. Based on the dependency rule CD1 in Table 1, since the Redundant Switch class is being impacted, all of its attributes and methods are added to the slice and stored in S_1[PSD] on line 8. Since there are no other model fragments to consider on line 5, the loop exits with S_1[PSD] as shown by the highlighted parts in Fig. 6.

Lines 10-17 (Expansion Level 2): Up to this point, R : CD − SD has not been considered in the slicing. In this expansion level, we do use it. First, the level 2 expansion temporary result S_2[PSD] is initialized on line 10 to the empty set. The outer iteration (lines 11-17) is over the model fragments from the level 1 expansion. We first consider the CD. On the opposite end of R : CD − SD is the PowerSlidingDoor : SD (which is M' in the algorithm on line 13). On line 14, we trace through R : CD − SD and add to $S_2[M']$ all the atoms related to those highlighted in the CD. This includes the Redundant Switch object and lifeline and all messages (or parts of them) that are traced back to attributes/methods of the Redundant Switch class in the CD. The result is added to S_2[PSD] on line 15 and can be seen in the highlighted parts of the SD in Fig. 7. Since no other model fragments exist in S_1[PSD] on line 11, the loop exits.

Line 18: The combined result S[PSD] is computed by computing the union of the results of the level 1 and level 2 slices, and can be seen as the result of the 1st iteration of the algorithm in the highlighted parts of Fig. 7.

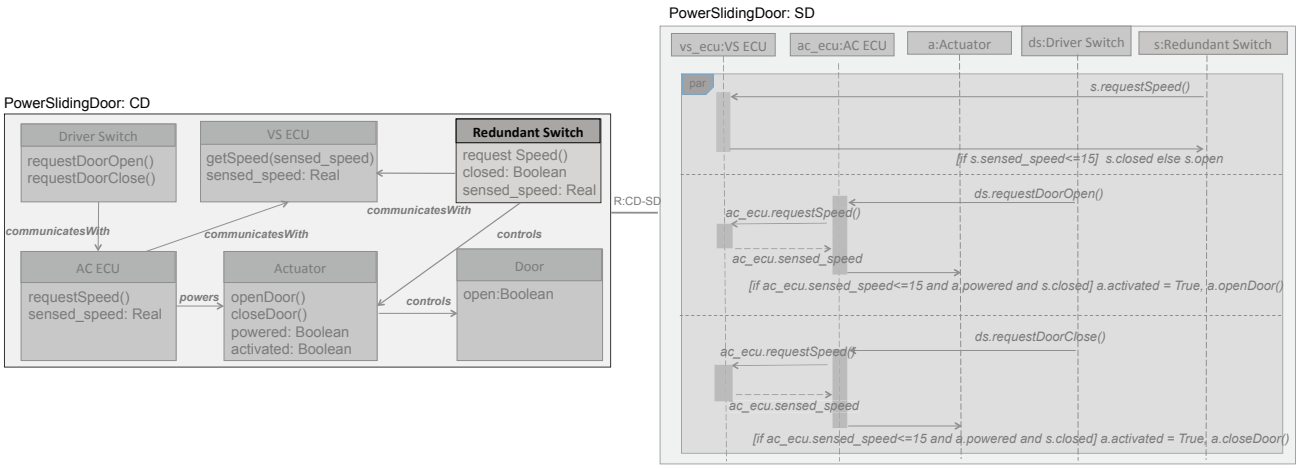


Figure 5: Slicing criterion S_c[PSD].

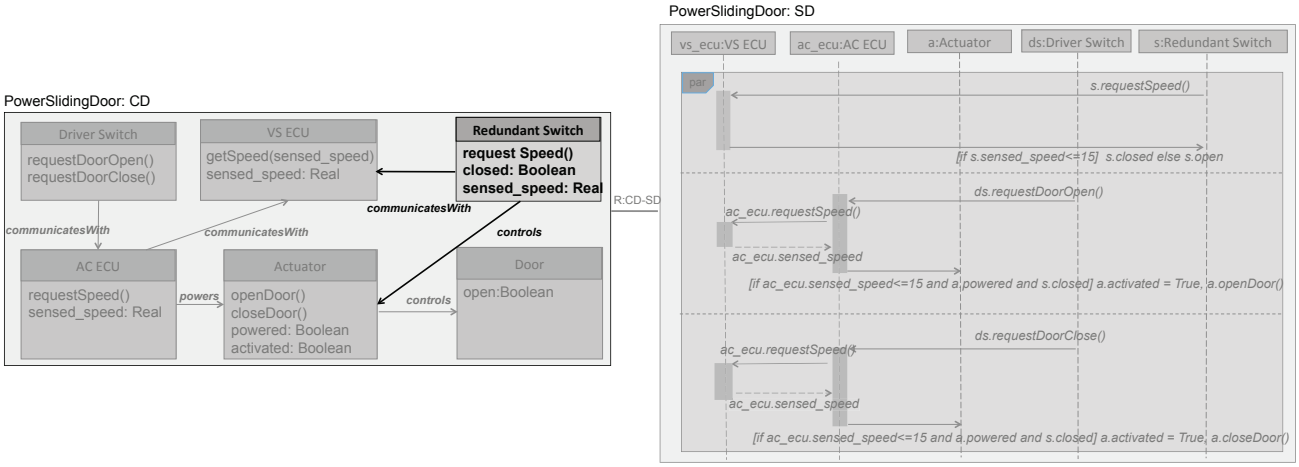


Figure 6: Result of level 1 slicing in 1st iteration.

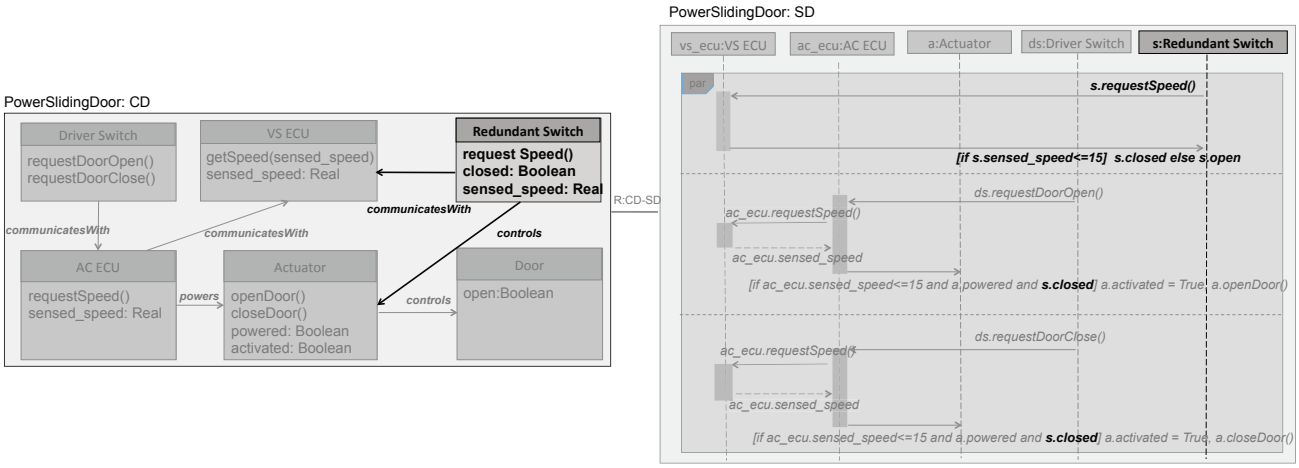


Figure 7: Result of the 1st iteration.

Line 19: In this line, we check to see if any actual expansion has occurred. Since the condition is not met (i.e., the result of the 1st iteration did indeed expand on the initial criterion) we iterate one more time.

2nd iteration of the outer loop (lines 2-19):

The slicing criterion S[PSD] in this iteration is the result of the previous iteration shown in Fig. 7. S_1[PSD] is reset again to the empty set.

Lines 4-9 (Expansion Level 1): First the CD is selected on line 5. Since none of the slicing dependency rules given in Table 1 apply, nothing is added to S_1[PSD] on line 8. Next, the SD is selected on line 5. Now, SD1 − SD3 rules for the SD slicer in Table 1 apply, and the SD slice is expanded to include the arrows of the top two messages and the entire expressions (and therefore messages and arrows) that the term s.closed appears in. This is seen in the highlighted parts of the SD portion of Fig. 8[1].

Lines 10-17 (Expansion Level 2): In this level, tracing across the R : CD − SD relationship from the CD to the SD (recall this is a unidirectional traceability relationship), since no new elements are introduced in the CD slice, nothing is traced to them in the SD. The result is an empty set.

Line 18: The results of the level 1 and the level 2 expansions are unioned and are reflected in the highlighted parts of Fig. 8.

Line 19: Since an expansion (w.r.t. the initial slice for this iteration) has occurred, the condition does not hold, and we iterate one more time on the outer loop.

3rd iteration of the outer loop (lines 2-19):

In this iteration, neither the CD nor the SD are expanded in the first level expansion as none of the dependency rules for their respective slicers holds. Similarly, no new elements are added, and therefore going through the trace links does not identify any other elements to be added to the expansion in level 2. The condition on line 19 now holds (no expansion has occurred), and the main loop of the algorithm exits.

Line 20 (Return): The current slice, S[PSD], which is shown in the highlighted parts of Fig. 8, is returned as the final result of the algorithm.

5.3 Post-processing

As suggested in Sec. 4, we perform a post-processing step, we expand the result of slicing algorithm shown in Fig. 8 to ensure the model fragments are well-formed and contextual information for referential integrity is included.

For the CD, the VS ECU and Actuator classes are included since both endpoints of associations communicatesWith and controls are needed for well-formedness.

For the SD, the VS ECU, AC ECU and Actuator objects and their lifelines are included to satisfy the well-formedness constraint of arrows requiring their lifelines. Also, the execution bar on the leftmost lifeline is included, as both of its input and output arrows are included in the result of the slicing.

Finally, all the methods and attributes of the Actuator class, as well as the sensed_speed attribute of the AC ECU class and the AC ECU class itself are added to satisfy the referential integrity condition between the SD and the CD (they are all referenced in the SD).

The detail of the final megamodel fragment produced after the slicing and post-processing is shown in the highlighted parts of Fig. 9. This can now be used to more efficiently complete the model evolution process by focusing only on the model parts impacted by the original deletion of Redundant Switch in the CD.

[1]Due to space limits, we have skipped visualizing the result at each step of the 2nd iteration and have shown the final result of the union only.

6. RELATED WORK

We identify three main categories of related work: work on model evolution, work on megamodeling operators, and finally, work on model slicing. We describe them below.

Model evolution. A survey on supporting the evolution of UML models in model-driven software development is presented in [14]. The scenarios that cause a model to change are discussed; these form the basis for megamodel evolution in our approach. In [22], the authors discuss some of the key problems of evolution in MDE, summarize the key state-of-the-art, and present some new challenges in research in this area. The problem of model evolution with respect to megamodels is stated as a "dependency heterogeneity" challenge. The authors express the need for a sound, precise theory of heterogeneous dependencies between MDE artefacts, as well as compliant and pragmatic tool support, both of which are complimentary to and/or are part of our current work.

Megamodeling operators. A formal approach to megamodeling, called *Mapping-Aware Megamodeling*, is presented in [9]. Our notion of a megamodel is consistent with it. The approach also describes category theory-based operations on the mapping-aware megamodels, but does not address megamodel slicing. In previous work [24], we presented a set of operators (Map, Filter, Reduce) that can be applied at the megamodel level. We are not aware of any other work in the area of applying operators at the megamodel level, and specifically, we have not seen any work addressing slicing of megamodels.

Model Slicing. We divide this area into work on *specific* model slicers, work on *generic* model slicers and work on slicing *multiple* models.

Specific Model Slicers. Numerous approaches have appeared in the literature describing slicers for specific model types. For example, [13] defines context-free model slicing and presents an algorithm for computing slices on UML class models. [18] also considers UML models, namely, class diagrams, individual state machines, and communicating sets of state machines. The approach achieves slicing of these models using model transformations. An approach for slicing state-based models, in particular, EFSM (extended finite state machine) models, is discussed in [16]. Finally, [17] proposes a slicing technique for UML architectural models, and demonstrates the uses of slicing for different purposes such as regression testing and understanding large architectures. Many other approaches (e.g., [21], [18]) are presented in the literature and can all be used as part of our framework as specific

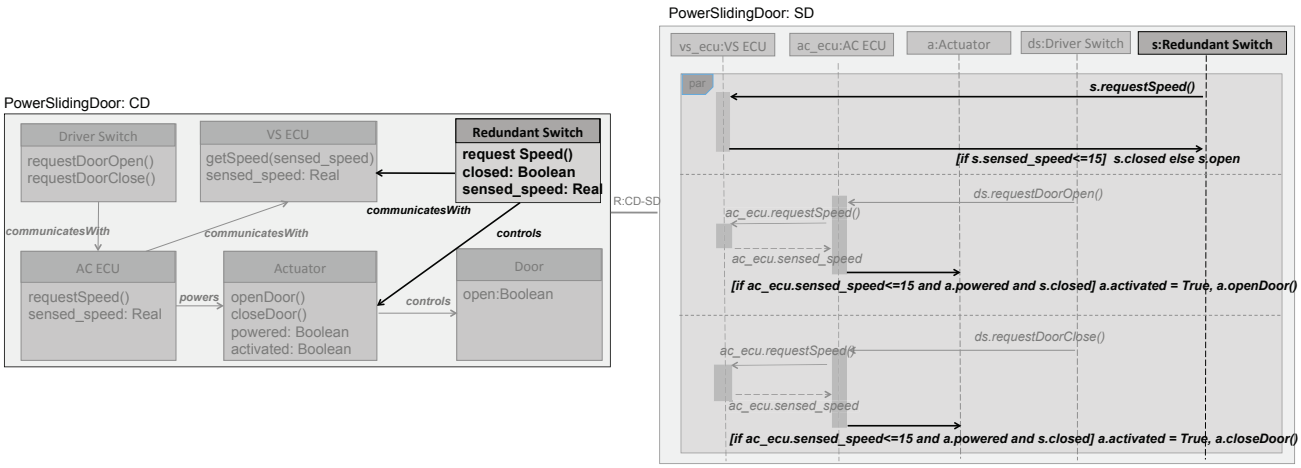


Figure 8: Result of 2nd iteration.

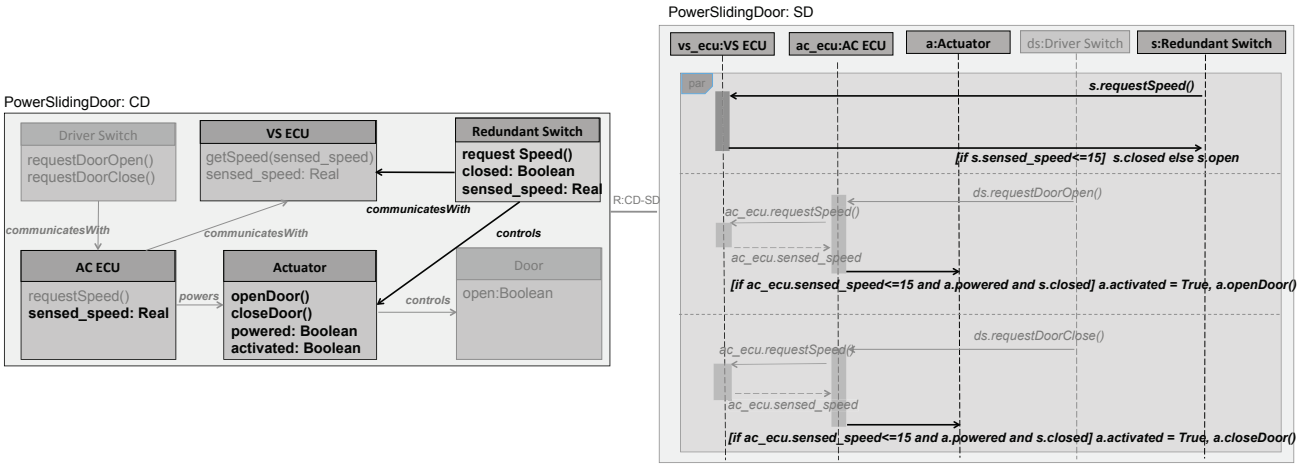


Figure 9: Output of algorithm after post-processing.

model type slicers for each of the model types in our heterogeneous megamodels.

Generic Model Slicers. Generic model slicing has also been studied in the MDE community. For example, the major contribution of [4, 5] is the Kompren language, which provides a generic approach to define a model slicer for a any domain-specific metamodel. The approach permits developers to either use "strict slicers" that output models which conform to their expected metamodel, or to define "soft slicers" that can output nonconforming models or even outputs that are not models. Although Kompren can be used for identifying specific type slicers in our framework, it is not applicable for megamodel slicing, where a megamodel slicer has to carefully invoke the specific type slicers. The work in [7] defines slicing at a theoretical level, whereas we focus on a more pragmatic approach. Also, the same work focuses on dynamic slicing, as does the transformation slicing work in [25], whereas our approach is considered a static slicing approach. As far as we know, none of the approaches in this category directly address megamodel slicing (whether the megamodels are heterogeneous or not).

Slicing Multiple Models. Although the work presented in [7] does not primarily focus on megamodel slicing, it briefly discusses heterogeneous slicing as the union of individual slicers. A slicing theory is presented at a high level and does not go into the details of implementing a megamodel slicing algorithm. From the modeling and safety community, [20] proposes a batch model slicer for slicing SysML models related to safety requirements. [10] presents a prototype tool called SafeSlice which performs the slicing needed in [20]. This line of work performs slicing on specific model types, whereas our work is a generic slicing approach. Also, the presented approach is amorphous slicing, where the result of the slice is not a model fragment of the original system. For example, transitions are added to sliced state-machines in order to preserve their behaviour. Our current approach only considers slices to be fragments of the original model (non-amorphous); however, we do plan to look at amorphous slicing in future work.

7. CONCLUSION

Model slicing is a useful technique for assessing change impact during model evolution activities. Although slicing of individual models has been investigated, slicing of heterogeneous model collections has received much less attention. In this paper, we have proposed a general algorithm for

slicing of heterogeneous model collections represented using megamodels and illustrated the algorithm on an automotive example. We analyzed the algorithm and showed that it behaves as expected with respect to termination, correctness, time complexity and minimality. Finally, we discussed the issues concerning slice well-formedness and referential integrity as well as how to generalize the algorithm to support arbitrary relationship types, N-ary relationship and nested megamodels. We are currently developing tooling for the algorithm using the Model Management INTeractive (MMINT) framework [8] and plan to use it to conduct more extensive case studies to better understand the strengths and weaknesses of the approach.

8. ACKNOWLEDGMENTS

This work is being done as part of the NECSIS project (www.necsis.ca), funded by Automotive Partnership Canada and NSERC.

9. REFERENCES

[1] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

[2] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proc. of CIDR'03*, volume 2003, pages 209–220, 2003.

[3] J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. In *Proc. of OOPSLA/GPCE Workshops*, 2004.

[4] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling Model Slicers. In *Proc. of MODELS'11*, pages 62–76. Springer, 2011.

[5] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and Generating Model Slicers. *SoSyM*, 14(1):321–337, 2015.

[6] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A Manifesto for Model Merging. In *Proc. of GAMMA@ICSE'06*, pages 5–12. ACM, 2006.

[7] T. Clark. A General Model-Based Slicing Framework. In *Proc. of Wrksp on Composition and Evolution of Model Transformations*, 2011.

[8] A. Di Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik. MMINT: A Graphical Tool for Interactive Model Management. In *Proc. of MODELS'15 (demo track)*, 2015.

[9] Z. Diskin, S. Kokaly, and T. Maibaum. Mapping-Aware Megamodeling: Design Patterns and Laws. In *Proc. of SLE'13*, pages 322–343, 2013.

[10] D. Falessi, S. Nejati, M. Sabetzadeh, L. Briand, and A. Messina. SafeSlice: A Model Slicing and Design Safety Inspection Tool for SysML. In *Proc. of ESEC/FSE'11*, pages 460–463. ACM, 2011.

[11] O. Gotel and A. Finkelstein. Contribution structures. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 100–107. IEEE, 1995.

[12] International Organization for Standardization. *ISO 26262: Road Vehicles – Functional Safety*, 2011. 1st version.

[13] H. Kagdi, J. I. Maletic, and A. Sutton. Context-Free Slicing of UML Class Models. In *Proc. of ICSM'05*, pages 635–638. IEEE, 2005.

[14] A. Khalil and J. Dingel. Supporting the Evolution of UML Models in Model Driven Software Development: a Survey. Technical Report 602, School of Computing, Queen's University, Ontario, Canada, 2013.

[15] S. Kokaly, R. Salay, V. Cassano, T. Maibaum, and M. Chechik. A Model Management Approach for Assurance Case Reuse due to System Evolution. In *Proc. of MODELS'16*, 2016. (to appear).

[16] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of State-Based Models. In *Proc. of ICSM'03*, pages 34–43. IEEE, 2003.

[17] J. T. Lallchandani and R. Mall. A Dynamic Slicing Technique for UML Architectural Models. *IEEE TSE*, 37(6):737–771, 2011.

[18] K. Lano and S. Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In *Proc. of MODELS'10*, pages 228–242. Springer, 2010.

[19] B. Li, X. Sun, H. Leung, and S. Zhang. A Survey of Code-Based Change Impact Analysis Techniques. *J. Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

[20] S. Nejati, M. Sabetzadeh, D. Falessi, L. Briand, and T. Coq. A SysML-based Approach to Traceability Management and Design Slicing in Support of Safety Certification: Framework, Tool Support, and Case Studies. *Information and Software Technology*, 54(6):569–590, 2012.

[21] K. Noda, T. Kobayashi, K. Agusa, and S. Yamamoto. Sequence Diagram Slicing. In *Proc. of APSEC'09*, pages 291–298. IEEE, 2009.

[22] R. F. Paige, N. Matragkas, and L. M. Rose. Evolving Models in Model-Driven Engineering: State-of-the-art and Future Challenges. *J. of Systems and Software*, 111:272–280, 2016.

[23] R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik. Lifting Model Transformations to Product Lines. In *Proc. of ICSE'14*, pages 117–128. ACM, 2014.

[24] R. Salay, S. Kokaly, A. Di Sandro, and M. Chechik. Enriching Megamodel Management with Collection-Based Operators. In *Proc. of MODELS'15*, pages 236–245, 2015.

[25] Z. Ujhelyi, Á. Horváth, and D. Varró. Towards dynamic backward slicing of model transformations. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 404–407. IEEE, 2011.

[26] M. Weiser. Program Slicing. In *Proc. of ICSE'81*, pages 439–449. IEEE Press, 1981.

Evolving Multi-Tenant SaaS Cloud Applications Using Model-Driven Engineering

Assylbek Jumagaliyev
School of Computing and Communications
Lancaster University
United Kingdom
a.jumagaliyev@lancaster.ac.uk

Jon Whittle
School of Computing and Communications
Lancaster University
United Kingdom
j.n.whittle@lancaster.ac.uk

Yehia Elkhatib
School of Computing and Communications
Lancaster University
United Kingdom
y.elkhatib@lancaster.ac.uk

ABSTRACT

Cloud computing promotes multi-tenancy for efficient resource utilization by sharing hardware and software infrastructure among multiple clients. Multi-tenant applications running on a cloud infrastructure are provided to clients as Software-as-a-Service (SaaS) over the network. Despite its benefits, multi-tenancy introduces additional challenges, such as partitioning, extensibility, and customizability during the application development. Over time, after the application deployment, new requirements of clients and changes in business environment result application evolution. As the application evolves, its complexity also increases. In multi-tenancy, evolution demanded by individual clients should not affect availability, security, and performance of the application for other clients. Thus, the multi-tenancy concerns add more complexity by causing variability in design decisions. Managing this complexity requires adequate approaches and tools. In this paper, we propose modeling techniques from software product lines (SPL) and model-driven engineering (MDE) to manage variability and support evolution of multi-tenant applications and their requirements. Specifically, SPL was applied to define technological and conceptual variabilities during the application design, where MDE was suggested to manage these variabilities. We also present a process of how MDE can address evolution of multi-tenant applications using variability models.

Keywords

Evolution; multi-tenancy; variability; cloud computing; cloud application; software product lines; model-driven engineering

1. INTRODUCTION

Cloud computing provides on-demand, scalable, and flexible computing resources to develop and deploy cloud applications [1]. Applications deployed on cloud are provided to clients as services over the Internet and are known as SaaS. As mentioned in [2], one key attribute of SaaS is multi-tenant efficiency, which enables economies of scale and efficient resource utilization by sharing a cloud infrastructure across multiple clients (i.e., tenants). A tenant is an organization or company with its end users that uses SaaS application.

As illustrated in Figure 1, there are generally two multi-tenancy patterns [3]: multiple instances multi-tenancy and single instance multi-tenancy. In the former, each tenant has a dedicated application instance on a shared hardware, operating system, or middleware. In the latter, tenants are served by a single application instance that runs on shared hardware and software infrastructure. We explore and address challenges that relate to the latter multi-tenancy pattern where tenants require isolation in application and database. Tenants may also want to extend or customize a business process workflow to cater for their specific needs. However, extensions and customizations of individual tenants should not affect the use of the application by other tenants. Thus, partitioning, extensibility, and customizability challenges emerge during the application development.

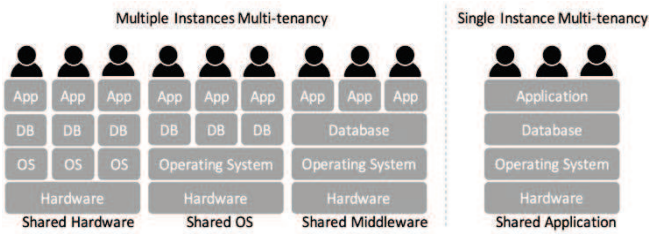


Figure 1. Multi-tenancy patterns

Over time, applications evolve because of changes in tenant requirements or new tenant requirements [6]. The evolution may imply changes in the application structure. Usually, cloud applications consist of several layers (e.g., presentation layer, data logic layer, and business logic layer) and changes in any layer may entail changes in other layers. Moreover, multi-tenancy requires the following architectural considerations to be addressed. First, the application layers must be multi-tenant aware to ensure tenant isolation. Second, the application must allow per tenant customization. Finally, each layer must scale independently of each other.

Cloud providers offer various technologies and tools for cloud application development. Nevertheless, multi-tenancy concerns cause additional variability challenges in design decisions such as different multi-tenant data architectures, partitioning schemas and design patterns. The variability represents different available options to implement a certain functionality and it should be considered in the whole lifecycle of multi-tenant applications to meet tenant requirements, and to leverage resource pooling and scalability of the cloud.

Variability can be efficiently managed using SPL techniques. Mainly, SPL engineering focuses on the development of software products from reusable core assets [7]. In SPL, software systems share common functionality, but each software system has some variable functionality [5].

Modeling the variability can also help to efficiently evolve applications. During the application development a set of variability models can be chosen for a given cloud deployment. When the application evolves, it is possible to evolve the corresponding code by selecting another set of options from the

variability model. For example, a multi-tenant data architecture can be modeled in different ways: 1) single database shared by all tenants, 2) a separate database for each tenant, or 3) multiple database instances where each instance serves a group of tenants. Initially, the developers might select a single database for all tenants. However, the security requirements of tenants may require a more isolated approach that cannot be provided in a single database instance. Therefore, the developer selects another multi-tenant architecture and the application evolves to multiple database instances.

The main contribution of our ongoing research is exploring combination of SPL and MDE techniques for managing variability in design decisions and evolving multi-tenant cloud applications. Others have advocated the integration of SPL and MDE for managing variability in multi-tenant cloud applications. For example, in [10], Orthogonal Variability Model (OVM) and Service Oriented Modeling Language (SoaML) were used to model variability and customizability in cloud applications. While in [4], a framework was proposed to model customizable multi-tenant cloud applications and to support their evolution. However, these approaches address application variability, customizability, and limited evolution scenarios, such as onboarding new tenants, removing tenants, and tenant customizations. In our approach, we use SPL to identify technological and conceptual variability prior to application implementation, where MDE concepts are applied to manage variability. Subsequently, variability models may efficiently support evolution of applications and their requirements. Moreover, we illustrate our approach by a multi-tenant application example.

The reminder of the paper is structured as follows. Section 2 describes variability in multi-tenant applications and their evolution. It also describes SPL and discusses related work in the field. Section 3 explains our approach for addressing variability and evolution challenges in multi-tenant applications. Section 4 presents a case study to motivate and illustrate our work. Finally, Section 5 concludes the presented approach.

2. BACKGROUND

In this section, we briefly explain variability in multi-tenant applications and their evolution. We also describe SPL and give an overview of related work.

2.1 Variability

Variability emerges in all levels of cloud applications. Abu-Matar et al. [4] categorized the variability into the following levels: application variability, business process variability, platform variability, provisioning variability, deployment variability and provider variability. Through this paper, we consider application variability and business process variability.

In application variability, different tenants may have different functional and non-functional requirements in addition to the core application. In business process variability, tenants may have varying business workflows. Therefore, the application must enable configuration and customization to meet tenant's goals and requirements. In [8], variability is separated as customer-driven variability and realization-driven variability. The customer-driven variability comprises tenant requirements. We can classify application and business process variability as customer-driven variability. The realization-driven variability represents different implementation options derived by customer-driven variability. In this paper, we use design decision variability as realization-driven variability.

2.2 Evolution

Evolution is an inevitable process in any software system [6] and multi-tenant applications are no exception. There are several reasons that trigger application evolution, such as fixing bugs, changes in business environment, improving security and reliability, changes in tenant requirements, or new tenant requirements. Applications should respond to such changes to maintain tenant satisfaction. In application level multi-tenancy, changes must be adapted at runtime without affecting availability, security, and performance of an application for other tenants. A key problem is implementing and managing required changes in applications [6].

2.3 SPL

SPL is a software engineering approach that focuses on the development of software products from reusable core assets [7]. It promotes feature modeling to analyze and identify the commonality and variability in applications [5]. Features are specific characteristics of an application and are classified in terms of capabilities, domain technologies, and implementation techniques [7]. Capabilities represent functional and non-functional characteristics that are provided by an application to clients. Domain technologies describe how to implement features regarding an underlying domain, where implementation techniques comprise commonly used generic approaches in the development. Features are also grouped as mandatory, optional, alternative and at-least-one-of (OR). Common features are mandatory features, while variability features may be optional, alternative or at-least-one-of. Optional features can be selected or neglected, only one feature must be selected from alternative features, and one or more features can be selected from at-least-one-of features.

2.4 Related work

Several authors have proposed using SPL or MDE techniques for managing variability in cloud applications to address multi-tenancy concerns. Moreover, there are some tools and frameworks for deploying, provisioning or supporting portability of cloud applications. However, none combined the strength of these two paradigms to address the multi-tenancy challenges, design decision variability challenges and evolution complexity.

2.4.1 MDE and SPLs

Mietzner et al. [8] proposed variability management in multi-tenant SaaS applications and their requirements using explicit variability models of SPL. Initially, the customer-driven variability and realization-driven variability were modeled using Orthogonal Variability Model (OVM). Then, the model was used to support customizability in applications. The authors also supported efficient SaaS applications deployment for new tenants based on the information about already deployed SaaS applications. Nevertheless, this approach addresses the application variability and does not support evolution.

Service line engineering (SLE) [9] (i.e., combination of service-oriented development and SPL) was introduced for customizable multi-tenant SaaS application development. SLE uses feature modeling to address engineering complexity and manage variability caused by application-level multi-tenancy. The main departure from SPL is that customizations are applied to a single application instance that is shared across multiple tenants. The author emphasized that SLE also supports application evolution.

Kumara et al. [11] described an approach for realizing service-based multi-tenant applications. This approach is also feature-

oriented as SLE and it supports evolution by enabling runtime sharing and tenant-specific variations using Dynamic SPLs.

CloudML [12], CAML [13], and CloudDSL [14] are examples of modeling languages for cloud applications that exploited MDE techniques. CloudML automates provisioning for cloud applications that run on multiple clouds. CloudDSL supports portability of applications by describing cloud platform entities, whereas CAML supports deployment and enables migration of existing applications to cloud. However, none of these modeling languages addresses multi-tenancy in design decisions or evolution of applications.

2.4.2 Combining MDE and SPLs

Shahin [10] integrated SPL and MDE to model variability for customizable SaaS applications. In this approach, SoaML was extended to model variability in all layers of Service Oriented Architecture (SOA). OVM from SPL was exploited to model variability as separate models. These separate models were used to generate a customization model for SaaS applications.

Cavalcante et al. [15] applied feature modeling to manage commonality and variability in cloud applications. In addition, they modeled costs regarding the use of cloud resources to minimize expenditure. They also used UML class diagram for features to identify dependencies.

Abu-Matar et al. [4] described a framework for modeling service-oriented customizable multi-tenant cloud applications. They exploited SPL for managing variability in services from multiple views (i.e., service-oriented views and cloud views). They also applied MDE for modeling multi-tenant aware application artifacts. In [17], the framework was complemented to support some evolution scenarios such as onboarding new tenants and removing tenants. In our approach, we address multi-tenancy concerns by modeling variability in design decisions that emerges during the architecting process. Thus, developer can use variability models for further support throughout the whole lifecycle of multi-tenant cloud applications.

3. OUR APPROACH

We consider an integration of feature modeling concepts and MDE techniques to address the design decision variability and evolution complexity in multi-tenant cloud applications. Our approach is based on the work of Jayaraman et al. [16]. The main idea of this approach is maintaining feature separation and detection of structural dependencies and conflicts between features during analysis and design modeling. Features or groups of features are modeled using UML, and a model composition language, MATA (Modeling Aspects using a Transformation Approach), detects relationships and conflicts. However, this approach requires additional work to support cloud application development and multi-tenancy.

Figure 2 illustrates modeling multi-tenant applications that consists of the following steps. Initially, common and variable functional and non-functional features with dependencies are captured using feature modeling. This helps to define available implementation options for the design decisions. Next, common features are used to model the core of the application using an UML composition language. Each variant feature is modeled in the MATA language with dependencies to the core UML model and relations to other features. This allows features to be modeled independently of each other and enables reuse of models. Further, a composed UML model is generated from the core UML model and selected models from models of variant features. At this stage,

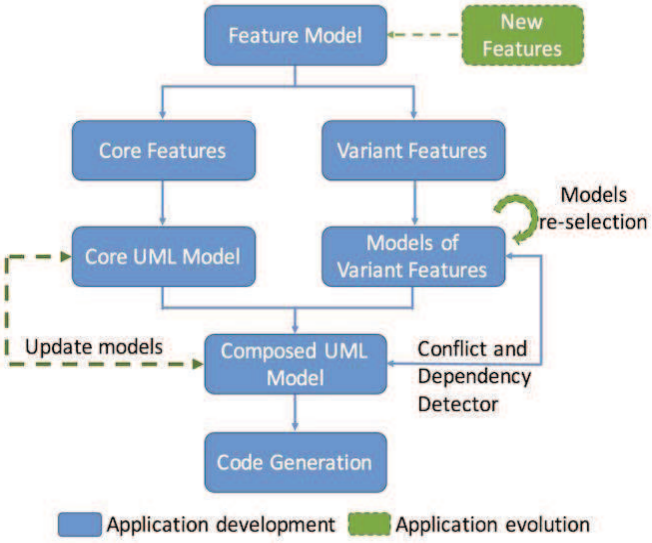


Figure 2. Multi-tenant application development and evolution with MATA

conflicts and dependencies of models are checked. Finally, source code specific to a particular cloud platform is generated.

Figure 2 also describes application evolution which may require models re-selection, adding new features, or a combination of both. In the case of model re-selection, developers pick appropriate features from the models of variant features. When evolution demands adding new features, developers identify whether new features are common or variable. The new common features affect the existing core UML model, whereas for each variable feature a corresponding model of variant feature is created. There might be cases when all new features are common or variable. In the former, only the core UML model is updated. While in the latter, new models are added to the models of feature variants and it requires models re-selection. Then, developers generate a composed UML model and source code.

4. CASE STUDY

To explore our approach, we present a Surveys service [2] case study by Microsoft. *Surveys* is a multi-tenant SaaS application for creating and managing online surveys. Tenants can create, publish surveys, and analyze results. Three different actors interact with the application: the application provider administrator, the tenant administrator, and the survey respondent. The application provider administrator manages all tenants and their surveys, whereas the tenant administrator manages its own surveys and survey results, and the survey respondent completes surveys.

Although multiple tenants use the same application instance with core functionalities and user interface layouts, each tenant can view and edit its own data. In addition, the application allows tenants to apply user interface customization by uploading their corporate logo, adding tenant name, welcome text, and contact details. Besides, tenants can customize the business process by choosing a standard or premium subscription type. With standard subscription, tenants can publish a limited number of surveys and cannot export their survey results. Premium subscription tenants can create and publish any number of surveys, export survey results for further analysis, and their requests are prioritized by the application.

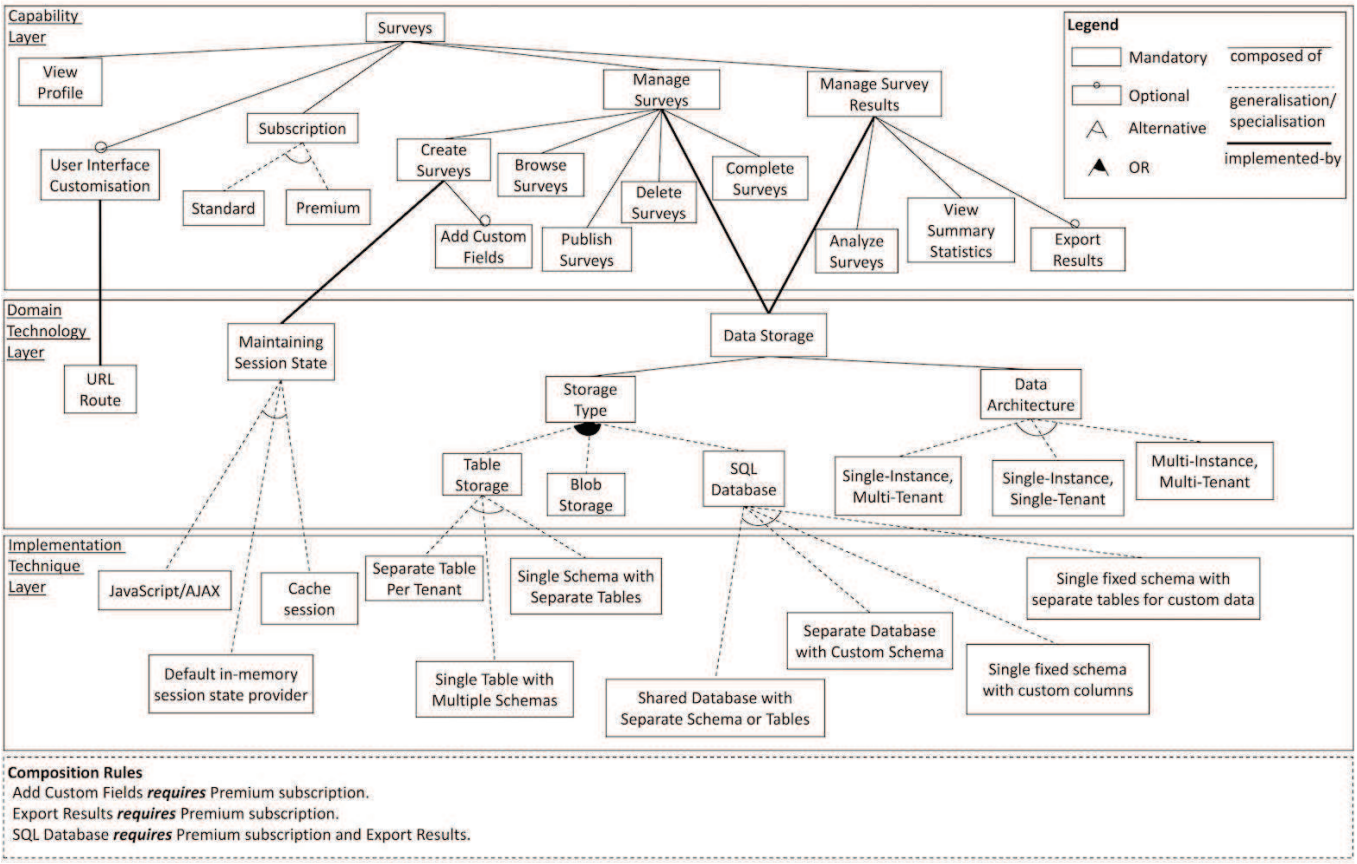


Figure 3. The feature model of the Surveys application.

4.1 Applying our Approach

As a first step, we constructed a feature model to define commonalities and potential variabilities in the application. An excerpt of the feature model is illustrated in Figure 3. As mentioned in Section 2.3, features were identified and categorized into three layers. The capability layer comprises the functional and non-functional features that are available for tenants. The domain technology layer describes the way of implementing features from the capability layer, and the implementation technique layer represents generic techniques to implement features on a cloud infrastructure. Further, the features were classified as mandatory, optional, alternative features, and at-least-one-of (OR). The mandatory features are common features that represent core components of the application that will always be present in any evolution of the cloud application. Whereas the optional, alternative and at-least-one-of features are variable features that describe different possible implementations. Once the common and variable features are defined, the process (as defined in Figure 2) would come up with a core UML model from the common features and models of variant features from the variable features. As a next step, a composed UML model from the core UML model and selected models of variant features would be generated.

Figure 3 shows that various options were modeled in the domain technologies and implementation techniques for realizing certain features. These variability models are used to support evolution. For example, the application uses a single database instance shared by all tenants. However, as the number of users per tenant increases, a more isolated approach must be selected from variability models to meet user requirements. With the MATA language multi-tenant data architectures are modeled separately with their dependencies to the core model and can be easily reused. Hence, developers can select any other multi-tenant data architecture model at any time during the application evolution.

4.2 Evolution Scenarios

Over the application lifetime, the functionality and quality of service offered by the application must increase to meet tenants' requirements. In this section, we consider some evolution scenarios that affect design decisions in the application structure.

When architecting the application structure, we decided to use a single database instance shared by all tenants. However, over time the number of tenants increases. Therefore, the number of concurrent end users and amounts of data stored by each tenant increase as well. Moreover, some tenants may require a separate database due to privacy requirements. These scenarios require a more isolated data storage approach and entail model re-selection from models of variant features. Thus, developers select either a single database instance for each tenant or multiple database instances for multiple tenants from the available data architecture models (as depicted in Figure 3).

For maintaining a session state while creating a new survey, we suggest JavaScript/AJAX technologies. This approach is simple, easy to maintain, scalable, and secure compare to other available implementation techniques under the Maintaining Session State feature. However, it relies on client-side JavaScript that makes it the least robust solution among available techniques. In the future,

to improve robustness and effectiveness, developers must decide between default in-memory session state provider and cache session. This scenario also requires model re-selection from existing models of variant features.

Another typical scenario is adding new features. For example, tenants may want to perform complex analysis on survey results. Currently, the application stores survey answers in blob storage. To provide the new feature, an SQL database (from different models under Storage Type) is the best solution for applying complex queries and join query. When adding a new feature, developers must identify whether the new feature is common or specific to certain clients. If the feature is common, the core UML model will be updated. If the feature is variable, the core UML model will remain the same and a model of variant feature for this variable feature will be generated. At this point, the MATA language detects relations and dependencies of the new feature to other features. The SQL Database also needs partitioning to support multi-tenancy. Thus, the developers must select one of the different partitioning models for SQL databases. Moreover, a new interface must be implemented to view and analyze survey data.

5. CONCLUSION

In this paper, we have proposed an integrated SPL and MDE modeling approach to address design decision variability and evolution concerns in multi-tenant SaaS cloud applications. We have applied feature modeling concepts to identify variability in implementation. The MATA language has been suggested to manage variability, and to support customization and evolution. Thus, the proposed approach allows features to be modeled independently. Furthermore, conflicts in the application structure and dependencies between models are detected. However, it requires improvements to enable cloud application development and multi-tenancy.

In our future work, we plan to enhance our approach by making the MATA language applicable for multi-tenant SaaS cloud applications and by developing a model to code transformation prototype to transform composed models to source code. A case study will be carried out to illustrate and evaluate the implemented tool. Moreover, we will compare our approach with other tools to identify benefits and drawbacks.

6. REFERENCES

[1] P. Mell, et al., "The NIST Definition of Cloud Computing", National Institute of Standards and Technology, Special Publication 800-145, Bethesda, Maryland, 2011

[2] D. Betts, et al., "Developing Multi-Tenant Applications for the Cloud on Windows Azure", Microsoft Patterns and Practices, 2013

[3] J. Guo, et al., "A framework for native multi-tenancy application development and management", Proceedings of the 9th IEEE Conference on E-Commerce Technology and the 4th IEEE Conference on Enterprise Computing, E-Commerce and E-Services, pp. 551–558, 2007.

[4] M. Abu-Matar, et al., "Towards Software Product Lines Based Cloud Architectures", Proceedings of the IEEE Conference on Cloud Engineering, 2014

[5] H. Gomaa, "Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures", Addison-Wesley Professional, 2004

[6] I. Sommerville, "Software Engineering", Pearson, 2010

[7] K. Lee, et al., "Concepts and guidelines of feature modeling for product line software engineering", Proceedings of the 7th Conference on Software Reuse: Methods, Techniques, and Tools, pp. 62–77, 2002.

[8] R. Mietzner, et al., "Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications", Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18-25, 2009

[9] S. Walraven, et al., "Efficient Customization of Multi-tenant Software-as-a-Service Applications with Service Lines", Journal of Systems and Software, Vol. 91, pp. 48-62, 2014.

[10] A. Shahin, A "Variability Modeling for Customizable SaaS Applications", International Journal of Computer Science and Information Technology, 6(5), pp. 39-49, 2014.

[11] I. Kumara, et al., "Sharing with a difference: Realizing service-based SaaS applications with run-time sharing and variation in dynamic software product lines", IEEE Conference on Services Computing, pp. 567–574, 2013

[12] A. Bergmayr, et al., "The Evolution of CloudML and its Manifestations", Proceeding of the 3rd Workshop on CloudMDE, 2015

[13] A. Bergmayr, et al., "UML-Based Cloud Application Modeling with Libraries, Profiles and Templates", Proceedings of the 2nd Workshop on CloudMDE, 2014.

[14] G. S. Silva, et al., "Cloud DSL: A Language for Supporting CloudPortability by Describing Cloud Entities", Proceedings of the 2nd Workshop on CloudMDE, 2014.

[15] E. Cavalcante, et al., "Exploiting Software Product Lines to Develop Cloud Computing Applications," the 16th Software Product Line Conference, 2012.

[16] P. Jayaraman, et al., "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis", Conference on Model Driven Engineering Languages and Systems, 2007

[17] F. Mohamed, et al., "SaaS Dynamic Evolution Based on Model-Driven Software Product Lines", Proceedings of the IEEE 6th Conference on Cloud Computing Technology and Science, 2014

Towards a Software Product Line for Machine Learning Workflows: Focus on Supporting Evolution

Cécile Camillieri
camillie@i3s.unice.fr

Luca Parisi
parisi@i3s.unice.fr

Mireille Blay-Fornarino
blay@i3s.unice.fr

Frédéric Precioso
precioso@i3s.unice.fr

Michel Riveill
riveill@i3s.unice.fr

Joël Cancela Vaz
joel.cancelavaz@gmail.com

Université Côte d'Azur, CNRS, I3S
Bat Templier, 930 Route des Colles
Sophia Antipolis, France

ABSTRACT

The purpose of the ROCKFlows project is to lay the foundations of a Software Product Line (SPL) that helps the construction of machine learning workflows. Based on her data and objectives, the end user, who is not necessarily an expert, should be presented with workflows that address her needs in the "best possible way". To make such a platform durable, data scientists should be able to integrate new algorithms that can be compared to existing ones in the system, thus allowing to grow the space of available solutions. While comparing the algorithms is challenging in itself, Machine Learning, as a constantly evolving, extremely complex and broad domain, requires the definition of specific and flexible evolution mechanisms. In this paper, we focus on mechanisms based on meta-modelling techniques to automatically enrich a SPL while ensuring its consistency.

Keywords

Software Product Line, Machine Learning Workflow, Evolution

1. INTRODUCTION

The answer to the question "What Machine Learning (ML) algorithm should I use?" is always "It depends." It depends on the size, quality, and nature of the data. It also depends on what we want to do with the answer [21].

The industry of cloud-based machine learning (*e.g.*, IBM's Watson Analytics, Amazon Machine Learning, Google's Prediction API) provides tools to learn "from your data" without having to worry about the cumbersome pre-processing and ML algorithms. To address such a challenge they propose fully automated solutions to some classical learning problems such as classification. Some other actors like Microsoft, with the Azure's Machine Learning platform, allow users to build much more complex ML workflows, in a graphical editor that is targeted towards ML experts.

The common point between these solutions is that they chose to select only a few algorithms, in comparison to the hundreds that are available. However data scientists know that the best algorithm will not be the same for each dataset [22]. Moreover, new algorithms are regularly proposed by data scientists for dealing with more or less specific problems and improving performances and accuracy [6]. Thus, in order to help users who want to build ML workflows, we have to propose a system that can present a large variety of algorithms to users, while helping them in their choices based on their data and objectives. At the same time, we should be able to extend the supported solutions at least by incorporating new algorithms. The challenge is to hide the complexity of the choices to the end user and to revise our knowledge with each addition: an algorithm can become less efficient compared to a new one, while the introduction of new pre-processing operations can extend the reach of algorithms already present.

The contribution of this paper is thus to describe a tool-supported approach, responding to this challenge: the ROCKFlows project[1].

The remainder of the paper is organized as follows. We discuss in the next section challenges we face and some related works. Section 3 describes the architecture that supports the project and two usage scenarios focusing each on a different user of ROCKFlows. We detail the evolution process and the correlated artefacts in Section 4. Section 5 concludes the paper and briefly discusses future work.

2. TOWARDS A SPL FOR MACHINE LEARNING WORKFLOWS

The purpose of the ROCKFlows project is to lay the foundations of a software platform that helps the construction of ML workflows. This task is highly complex because of the increasing number and variability of available algorithms and the difficulty in choosing the suitable and parametrized algorithms and their combinations. The problem is not only on choosing the proper algorithms, but the proper transformations to apply on the input data. It is a trade-off between many requirements (*e.g.*, accuracy, execution and training time).

Since Software Product Line (SPL) engineering is concerned with both variability and systematically reusing development assets in an application domain [5], we have based our project on SPL and model-driven techniques. The SPL engineering separates two processes: *domain engineering* for defining commonality and the variability of the product line and *application engineering* for deriving product line appli-

[1]ROCKFlows stands for Request your Own Convenient Knowledge Flows.

cations [15]. Similarly, the ROCKFlows project requires on one hand to build a consistent SPL, allowing end users to get reliable workflows, and on the other hand to allow evolution of this SPL to integrate new algorithms and pre-processing treatments.

Based on these requirements, we have identified the following challenges, addressing the needs for building and evolving a SPL in a domain as complex and changing as ML, ensuring a global consistency of the knowledge and scalability of the system.

C1: Exploratory project in a complex environment.

Making a selection among the high number of data mining algorithms is a real challenge: more than hundreds of algorithms exist that can tackle a single ML problem such as classification. While work exists to try and rank their performance [6], it only gives an overview of which algorithms are best in average, not for a given specific problem and not according to different pre-processing pipelines.

Data scientists often approach new problems with a set of best practices, acquired through experience. However, there is few scientific evidence as to why an algorithm or pre-processing technique leads to better results than another and in which case. Thus, one of the biggest challenge for this project is so to find a proper way to characterise algorithms and to compare them, relatively to the very broad spectrum of user needs and data representations.

Collaboration between SPL developers and data scientists induces a complex software ecosystem [13] where some mathematical results may or not find a correspondence at end user problem level. Heterogeneity of formalisms induces that new evolutions are regularly discussed between the different stakeholders.

Given these domain requirements, meta-model driven engineering provides an efficient and powerful solution to address the complexity of the ecosystem through support of separation of concerns and collaborations. In order to operationalize it, we chose to consider this environment as a set of components relying on different meta-models for which the evolution mechanisms are exposed through services.

C2: SPL building in a constantly evolving environment.

While the number of ML algorithms and techniques constantly grows, the fundamental understanding of ML internal mechanisms is not stable enough to allow us to set any knowledge in stone. Both the domain and our understanding of it evolve quickly, forcing constant evolution of the SPL.

The line evolves in particular through the addition of new algorithms and pre-processings. We run experiments to identify dataset patterns leading to similar behavior of algorithms on different concrete datasets. The high number of possible combinations (variability of compositions and algorithms) as well as the frequent changes in ML require evolution mechanisms that are both incremental and loosely coupled with the elements presented to the end user.

As of today, ROCKFlows' SPL contains roughly 300 features, and 5000 constraints, representing 70 different ML algorithms, 5 pre-processing workflows, and is mostly focused on classification problems.

Evolution in SPLs has been a challenge for many years [15]. In particular, several works exist on evolution of Feature Models (FM) [8, 1]. They propose different mechanisms for maintaining consistency of evolving FMs. In our case, we rely on these operations to update our models, but we had to encapsulate them in business oriented services.

Moreover, despite the huge variability of the system, we have decided to propose the end user only choices that can lead to a proper result. Hence, it should not be possible to build a configuration for which we would not be able to generate a workflow. For instance, it will not be possible for a user to select, relatively to a given dataset, a performance value for which we have no algorithm that can reach such requirements. It is necessary for us to ensure a consistent configuration process [20].

Contrary to the works allowing several users to modify a model in contradictory manners and aiming to reconcile those [3], here we are in a setting where only consistent evolutions are possible. Thus we did not have to handle co-evolution problems. Like the approach used in SPLEMMA [17], "Maintenance Services" define the semantics of evolution operations on the SPL ensuring its consistency. However, the analogy between meta-elements manipulated in ROCKFlows and SPLEMMA is hard to establish, especially because our solution and problem spaces are in a constant evolution. Thus the associated meta-models are not stable, implying intraspatial second degree evolutions [19] *i.e.*, several spaces and mappings are simultaneously modified; *e.g.*, Adding a non-functional property kind, due to some improvement of the experiment meta-model, involves to extend the FM and corresponding end user representation (problem space) and generation tools to take into account this new feature.

C3: User Centered SPL.

We identify three stakeholders, each leveraging challenges.
- **SPL users** are the end users of the SPL: it can be a neophyte who is looking for a solution to extract information from a dataset as well as an expert who wants to check or learn dependencies among dataset properties, algorithms, targeted platforms and user objectives. They want to use a system helping them to master the variability, *i.e.*, to express their requirements and get their envisioned ML workflow. Both of these do not know about FMs and may need complementary information like examples of uses of the algorithms, details about the algorithm author or implementation, etc. Thus, the visualization of a FM as in standard tool is not adapted. Creating a user interface dedicated to the SPL is also problematic knowing the changing nature of the system. At the same time, in a agile process, we need to test the SPL with users in order to align it with their needs, which are hard to identify a priori.

Some flowcharts have been designed to give users a bit of a rough guide on how to approach ML problems [16, 14]. ROCKFlows wants this approach to be operational. So, we do not aim for the construction of workflows by assembly, but the automatic production of these workflows, without a direct contribution of the user in this construction process. Works such as these are however potential targets for the generation of the workflows where proposed optimization could then be used automatically. Moreover, faced with the multitude of such systems (Clowdflows [10], MLbase [11], Weka [9]), it is right to allow the user to select her execution target(s) so that the production is limited only to the ML workflows implemented by these platforms.
- **External Developer** are domain experts who contribute

to the SPL. They do not have all the knowledge of the system and contribute by adding new algorithms. They have to be able to contribute separately with minimal interference.
- **Internal developers** are leaders of the SPL. They have the knowledge of the global architecture and manage contributions of external developers to integrate them. They have to be able to maintain the platform and to ensure the consistency of all products despite the evolution of the ecosystem.

3. ARCHITECTURE FOR ROCKFlows

3.1 ROCKFlows Big Picture

Figure 1 represents the current proposed architecture for ROCKFlows on the component level. On the left of the central vertical line are the components that will be necessary for the end user to configure her worfklow. On the right, components enabling users to add their own ML algorithm in the system is described. Relationships between the different components are relying on meta-models.

We now describe this architecture through the description of two scenarios:

3.1.1 Scenario 1: Configuration of a ML Workflow

A user willing to configure a new ML workflow will do so through a web-based configuration interface[2]. The process requires at least the following steps, as visible on the left of figure 1:

(a) The Graphical User Interface (`GUI`) requests display metadata on the Feature Model. Metadata associate each unique feature of the model with descriptions, references or other artefacts aiming to help non-experts in their choices. Figure 2 shows a screenshot of our GUI. Here, user is presented with the choice of its main objective, in the form of questions.

(b) Once the FM is loaded and displayed properly with the metadata, the user configures the underlying FM by responding to questions. The `Feature Model` component in the figure exposes a web-service that allows for configuration on any FM, through the use of SPLAR's API [12].

(c) Once a valid and complete configuration has been defined, it is sent to the `Workfow` component. The configuration is then transformed into a Platform Independent workflow model, that can in a second step be used to generate executable code for different target platforms.

(d) The Generator may require access to the base of algorithms handled by the system in order to be able to produce the proper code.

Though it is not described here, depending on user's preference, the generated workflow would either be provided to the user or directly executed by the target platform.

3.1.2 Scenario 2: Submission of a new algorithm

We will now focus on the introduction of new ML algorithms in the SPL. Such an action has impacts on several parts of the system:
- **SPL:** At least a new feature representing the algorithm should be added in the FM;
- **GUI**: Display metadata should be updated to reflect this new feature in the GUI;
- **Generation:** If we can execute the algorithm, the generator should be updated to allow it either through code or a reference to the corresponding element in target platform(s);
- **Experiments** should be made with this new algorithm in order to compare it to the other known algorithms. Currently, the properties that are considered are accuracy of the results, execution time of the workflow, and memory usage. If experiments can be achieved, *i.e.*, `Experiments` module has access to algorithm execution and results, the SPL needs to be updated with performance information for this algorithm but also for all the algorithms whose ranking has changed.

The central component `SPLConsistencyManager`'s role is to ensure that all required changes are made across the whole system. Through the present scenario, we describe how this component handles the impacts mentioned above.

(1) As an External Developer wants to add a new algorithm, the `GUI` presents her with proper information that she needs to provide. Because this information is meant to change as the system encompasses more possibilities of Machine Learning, it should be easy to change. Hence, the `SPLConsistencyManager` provides the GUI with the information needed to add a new algorithm in the tool. The GUI presents then a generated form to the user.

(2) Once the External Developer has provided all information on the algorithm, it is sent to the Consistency Manager and dispatched among the other components.

(3) If the External Developer provided code to execute the algorithm, the `Experiments` component is requested to run tests on the algorithms, in order to find its performance. This component is described more precisely in the next section.

(4) Once experiments are finished, the manager analyses the results to find whether any inconsistency was found between the information provided by the user and the results. If not, the algorithm can be added both to the Feature Model and base of currently supported algorithms.

(5) Finally, once the feature has been added to the FM, its display metadata can be filled with the information provided by the expert.

3.2 Component for experiments on algorithms

As we want even non-expert users to be able to get the appropriate algorithm for their need, we chose to express higher level goals such as *best* accuracy or *quickest* execution time in the FM. This representation allows to filter out a number of algorithms by offering users with trade-offs over these goals. In a second step, or a more advanced mode, using other models such as requirements engineering Goal Models is considered. In combination with our FM, it could allow during configuration to present the user with more precise expected values for her non functional goals [2].

To be able to express knowledge such as "best", "average" or "worst" accuracy, we need an appropriate way to compare algorithms on a given problem. This ranking among algorithms is computed by our `Experiment` module. The

[2]The interface is accessible at http://rockflows.i3s.unice.fr

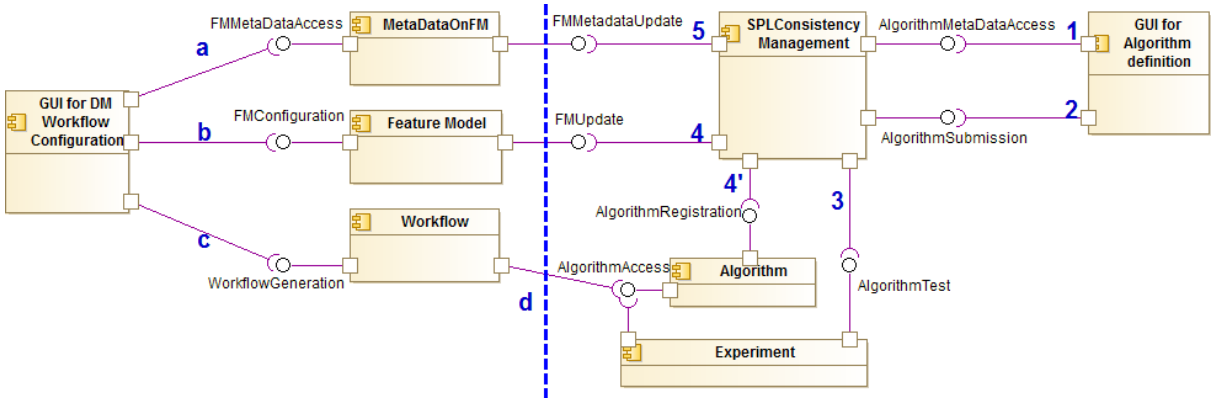


Figure 1: High level architecture for ROCKFlows.

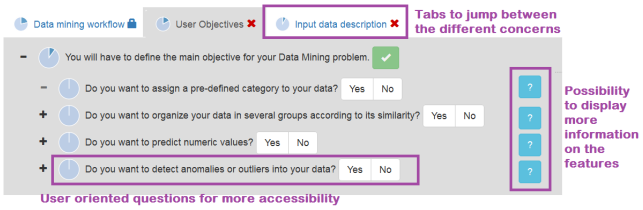


Figure 2: Screenshot of ROCKFlows' GUI.

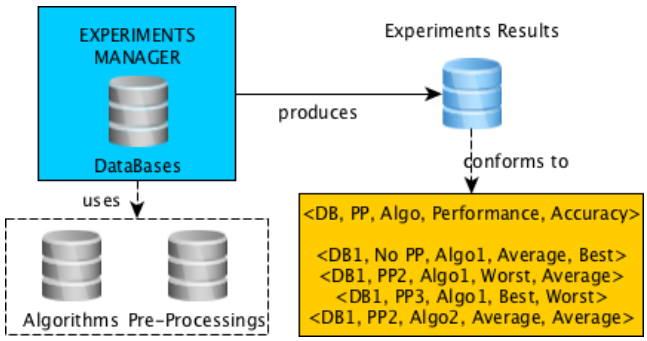


Figure 3: Experiments

component runs each known algorithm on the available compatible datasets and stores its performance on the different properties for each dataset. On top of that, it will transform the datasets with a set of available pre-processing operations and test again each algorithm with those new sets.

Algorithms results on similar datasets are then compared to one another to get a result similar to the one visible on figure 3. Though it will not be discussed here, we have defined an algorithm based on classical ML and statistical methods to pull off this comparison and regroup datasets in so-called dataset patterns. This knowledge can then be pushed in the FM in the form of constraints linking a functional objective, an algorithm, a dataset pattern and the ranking for each of the properties, also depending on the possible pre-processings for this dataset.

As presented in section 3.1.2, the `Experiment` component is driven by the `SPL Consistency Manager`, that is charged to start experiments, gather and validate results before incorporating them in the SPL. As new algorithms are added, all experiments do not have the need to be executed again, however the ranking of the algorithms must be updated to take in account the newest algorithm.

4. ARTEFACTS TO SUPPORT EVOLUTION

This section discusses how we allow users to provide information about new algorithms, how we use it to update the system, and how we can ensure consistency despite the multiple impacts of these changes.

4.1 Meta-models

Figure 4 shows an excerpt of the meta-elements that deal with evolution of the SPL. Each component described earlier corresponds to a meta-model, and the `SPLConsistencyManager` maintains consistency among them.

4.1.1 SPL: Feature Model and Configuration

Our feature model is represented in the SXFM (Simple XML Feature Model)[3]. The rest of our SPL handling is also made through SPLOT's FM reasoning library, SPLAR[4].

4.1.2 Addressing non-expert users

In order to make the system as accessible as possible, additional information on features must be set, such as descriptions or examples, as well as closed questions that will be asked to the end user during configuration. This metadata on the practical features is handled in a dedicated meta-model `AlgorithmDescriptionMM` and used to build the GUIs that are presented to the end users and external developers. The model is briefly described in subsection 4.2.

4.1.3 Handling the results of Experiments

Information such as the accuracy ranking of algorithms according to dataset patterns is managed by the `Experiment` component. The expected format of this data is designed in a dedicated meta-model `ExperimentPropertyMM`. It evolves as we gain knowledge and experience.

[3]http://ec2-52-32-1-180.us-west-2.compute.amazonaws.com:8080/SPLOT/sxfm.html

[4]An excerpt of the meta-model used for both FM and configuration definition in the library, can be found in https://github.com/FMTools/sxfm-ecore/blob/master/plugins/sxfm/model/sxfm.png

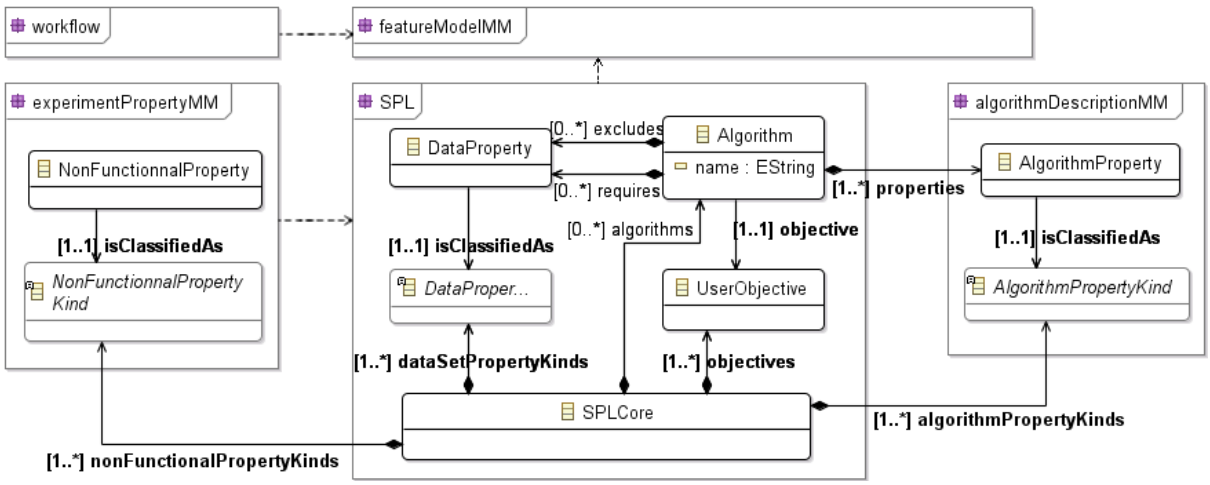


Figure 4: SPL Core Meta-models excerpt

4.2 Metadata on Algorithms

Data scientists adding their algorithms in the system need to provide at least:
- **The high level ML objective** the algorithm can be used for: classification of data, prediction of numerical values (regression), anomaly detection, etc.;
- **Properties** of the algorithm in regards to **input data**: which data types the algorithm supports, can the algorithm handle missing values in the data, etc.;
- **A description** of the algorithm, examples of its use, references to publications or web pages describing the algorithm. As described in section 4.1.2, those will be displayed in the configuration interface;
- If possible, **code** that will allow us to run the algorithm in our tool, so that we can both compare it to the others algorithms and provide executable workflows to end users.

Through the definition of theses elements in the `AlgorithmDescriptionMM` meta-model, a form is automatically generated and presented to the domain expert. Thus, the model can be extended to add new properties for the algorithms and will be automatically handled by the GUI. However the impact of such changes on the FM still has to be handled by the SPL manager. We do not know if tools such as the one described in [7] could help because those changes mostly impact code.

4.3 Domain driven tooled approach to manage Feature Model evolution

Even though we have defined a single FM for ROCK-Flows, we put a focus on separating concerns in it. The tree is currently separated in 4 sub-trees handling: input data description, user objectives, the processing algorithms, and expected properties of the generated workflow. This separation of concerns provides a first level of modularity for the model.

Linking Domain artefact and FM structure. Metadata external to the FM itself defines particular points in the FM where feature can be inserted. Only the sub-trees that need to be modified are considered. In our example, we add all algorithms responding to the classification problem in the same sub-tree. This mechanism enables us to extend the model cleanly, and abstract ourselves from the exact hierarchy of the features. So, adding new class of algorithms or modifying the structure of the FM can easily be achieved. Once the feature for the algorithm has been created, additional constraints need to be defined between the algorithm and other features, in particular those describing input data.

Generating domain constraints. Only certain types of constraints must be defined among those different sub-trees. For instance algorithms can define constraints towards input data, such as "SVM *implies* Numerical Data" but never the other way around. However such a constraint only applies in a workflow if no pre-processing is used. So, this constraint has to be transformed to express a constraint depending on the pre-processings that can be applied. The complexity of these cases, their multitude and the frequency of evolution lead us to encapsulate the generation of those constraints in dedicated operators, working on given ensembles (*e.g.*, pre-processing set that returns numerical Data). They also introduce features that are hidden to the end user, allowing us to tame this complexity.
It is interesting to note that, depending of the semantics associated to the features, different constraints should be generated. For instance, if an algorithm cannot deal with missing values, a constraint "algo *excludes* missing values" needs to be generated. In the other case, a constraint "algo *implies* missing values" should never be generated because such an algorithm can still be used even if no missing value is present in the input data. Like previously, this higher level knowledge of the features is defined in our metadata. It allows us to ensure that all necessary constraints are properly defined for all algorithms we add into the SPL.

5. CONCLUSION AND FUTURE WORK

In this paper, we have outlined some of the difficulties related to building and evolving a SPL for ML workflows. To handle the line's complexity and evolution, we have proposed an architecture organized around a set of meta-models and transformations encapsulated within services, A large number of complementary perspectives are considered, both on mechanisms to build the SPL and on the business approach of ML.

The complexity we are facing requires an agile and pragmatic approach in which users are given the opportunity

to provide feedback during the early stages of the project. The necessity for evolution of the system leads us today to consider moving towards approaches focused on the reuse of existing components [18]. This would allow for a necessary control over the system's evolution.

The domain of ML is currently booming with propositions for algorithms, distributivity of execution and the opening to a larger audience. We consider to extend our approach to integrate the necessary parameters for distributing workflows' execution as well as proposing deep learning workflows. A longer term question is to target Scientific Workflow Management systems, allowing so to explore data driven execution through transformations targeting the specification interchange language called WISP [4].

Finally, we wish to allow the end user to express her needs in business terms, through an approach similar to IBM Watson Analytics[5]. It is a question of integrating the state of the art practices in our modelisation, without losing the power of our evolutionary approach, driven by experiments.

6. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. Separation of Concerns in Feature Modeling: Support and Applications. In *AOSD'12*. ACM, 2012.

[2] O. Alam, J. Kienzle, and G. Mussbacher. *Concern-Oriented Software Design*, pages 604–621. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[3] A. Anwar, S. Ebersold, B. Coulette, M. Nassar, and A. Kriouile. A Rule-Driven Approach for composing Viewpoint-oriented Models. *Journal of Object Technology*, 9(2):89–114, 2010.

[4] B. F. Bastos, R. M. M. Braga, and A. T. A. Gomes. WISP: A pattern-based approach to the interchange of scientific workflow specifications. *Concurrency and Computation: Practice and Experience*, 2016.

[5] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[6] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? *Journal of Machine Learning Research*, 15:3133–3181, 2014.

[7] S. Getir, M. Rindt, and T. Kehrer. A generic framework for analyzing model co-evolution. In *Proceedings of the Workshop on Models and Evolution co-located with MoDELS 2014, Valencia, Spain, Sept 28, 2014.*, pages 12–21, 2014.

[8] J. Guo, Y. Wang, P. Trinidad, and D. Benavides. Consistency maintenance for evolving feature models. *Expert Systems with Applications*, 39(5):4987–4998, 2012.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[10] J. Kranjc, V. Podpečan, and N. Lavrač. Clowdflows: a cloud based scientific workflow platform. In *Machine Learning and Knowledge Discovery in Databases*, pages 816–819. Springer Berlin Heidelberg, 2012.

[11] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. J. Franklin, and M. Jordan. MLbase: A Distributed Machine-Learning System. In *CIDR*, 2013.

[12] M. M. Mendonça, M. Branco, D. Cowan, and M. Mendonca. S.P.L.O.T.: software product lines online tools. In *OOPSLA*, pages 761–762. ACM Press, 2009.

[13] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik. Studying Evolving Software Ecosystems based on Ecological Models. In *Evolving Software Systems*, pages 297–326. Springer, 2014.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, and al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. http://scikit-learn.org/stable/tutorial/machine_learning_map/.

[15] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[16] B. Rohrer. Machine learning algorithm cheat sheet for Microsoft Azure Machine Learning Studio. https://azure.microsoft.com/en-us/documentation/articles/machine-learning-algorithm-cheat-sheet/.

[17] D. Romero, S. Urli, C. Quinton, M. Blay-Fornarino, P. Collet, L. Duchien, and S. Mosser. SPLEMMA: a generic framework for controlled-evolution of software product lines. In *International Workshop on Model-driven Approaches in SPL (MAPLE)*, volume 2013, pages 59–66, 2013.

[18] M. Schöttle, O. Alam, J. Kienzle, and G. Mussbacher. On the modularization provided by concern-oriented reuse. In *Proceedings of MODULARITY'16*, pages 184–189, New York, NY, USA, 2016. ACM.

[19] C. Seidl, F. Heidenreich, U. Aßmann, and U. Aßmann. Co-evolution of Models and Feature Mapping in Software Product Lines. In *Proceedings of SPLC'12*, pages 76–85, New York, NY, USA, 2012. ACM.

[20] S. Urli, M. Blay-fornarino, and P. Collet. Handling Complex Configurations in Software Product Lines : a Tooled Approach. In ACM, editor, *Proceedings of SPLC'14*, pages 112–121, Florence, Italy, 2014.

[21] I. H. Witten, E. Frank, and M. a. Hall. *Data Mining: Practical Machine Learning Tools and Techniques (Google eBook)*. 2011.

[22] D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Comput.*, 8(7):1341–1390, Oct. 1996.

[5]https://www.ibm.com/analytics/watson-analytics/