Automated Software Engineering (2023) 30:17
https://doi.org/10.1007/s10515-023-00381-1

Mining domain-specific edit operations from model repositories with applications to semantic lifting of model differences and change profiling

Christof Tinnes[1,3] · Timo Kehrer[2,4] · Mitchell Joblin[1,3] · Uwe Hohenstein[1] · Andreas Biesdorf[1,5] · Sven Apel[1,6]

Received: 14 April 2022 / Accepted: 26 February 2023 / Published online: 26 April 2023

Abstract

Model transformations are central to model-driven software development. Applications of model transformations include creating models, handling model co-evolution, model merging, and understanding model evolution. In the past, various (semi-) automatic approaches to derive model transformations from meta-models or from examples have been proposed. These approaches require time-consuming handcrafting or the recording of concrete examples, or they are unable to derive complex transformations. We propose a novel *unsupervised* approach, called Ockham, which is able to learn edit operations from model histories in model repositories. Ockham is based on the idea that meaningful domain-specific edit operations are the ones that *compress* the model differences. It employs frequent subgraph mining to discover frequent structures in model difference graphs. We evaluate our approach in two controlled experiments and one real-world case study of a large-scale industrial model-driven architecture project in the railway domain. We found that our approach is able to discover frequent edit operations that have actually been applied before. Furthermore, Ockham is able to extract edit operations that are meaningful—in the sense of explaining model differences through the edit operations they comprise—to practitioners in an industrial setting. We also discuss use cases (i.e., semantic lifting of model differences and change profiles) for the discovered edit operations in this industrial setting. We find that the edit operations discovered by Ockham can be used to better understand and simulate the evolution of models.

Keywords Edit operations · Model-driven engineering · Software product line engineering · Model versioning

✉ Christof Tinnes
christof.tinnes@siemens.com

Extended author information available on the last page of the article

1 Introduction

Software and systems become increasingly complex. Various languages, methodologies, and paradigms have been developed to tackle this complexity. One widely-used methodology is model-driven engineering (MDE) (Rodrigues Da Silva 2015), which uses models as first class entities and facilitates generating documentation and (parts of the) source code from these models. Usually, domain-specific modeling languages are used and tailored to the specific needs of a domain. This reduces the cognitive distance between the domain experts and technical artifacts. A key ingredient of many tasks and activities in MDE are model transformations (Sendall and Kozaczynski 2003).

We are interested in edit operations as an important subclass of model transformations. An *edit operation* is an in-place model transformation and usually represents regular evolution (Van Deursen et al 2007) of models. For example, when moving a method from one class to another in a class diagram, also a sequence diagram that uses the method in message calls between object lifelines needs to be adjusted accordingly. To perform this in a single edit step, one can create an edit operation that executes the entire change, including all class and sequence diagram changes. Some tasks can even be completely automatized and reduced to the definition of edit operations: Edit operations are used for model repair, quick-fix generation, auto completion (Ohrndorf et al 2018; Hegedüs et al 2011; Kögel et al 2016), model editors (Taentzer et al 2007; Ehrig et al 2005), operation-based merging (Kögel et al 2009; Schmidt et al 2009), model refactoring (Mokaddem et al 2018; Arendt and Taentzer 2013), model optimization (Burdusel et al 2018), meta-model evolution and model co-evolution (Rose et al 2014; Arendt et al 2010; Herrmannsdoerfer et al 2010; Getir et al 2018; Kolovos et al 2010), semantic lifting of model differences (Kehrer et al 2011, 2012a; ben Fadhel et al 2012; Langer et al 2013; Khelladi et al 2016), model generation (Pietsch et al 2011), and many more.

In general, there are two main problems involved in the specification of edit operations or model transformations in general. Firstly, creating the necessary transformations for the task and the domain-specific modeling languages at hand using a dedicated transformation language requires a deep knowledge of the language's meta-model and the underlying paradigm of the transformation language. It might even be necessary to define project-specific edit operations, which causes a large overhead for many projects and tool providers (Kehrer et al 2017; Mokaddem et al 2018; Kappel et al 2012). Secondly, for some tasks, domain-specific transformations are a form of tacit knowledge (Polanyi 1958), and it will be hard for domain experts to externalize this knowledge.

As, on the one hand, model transformations play such a central role in MDE, but, on the other hand, it is not easy to specify them, attempts have been made to support their manual creation or even (semi-)automated generation. As for manual support, visual assistance tools (Avazpour et al 2015) and transformation languages derived from a modeling language's concrete syntax (Acreţoaie et al 2018; Hölldobler et al 2015) have been proposed to release domain experts from the need of stepping into the details of meta-models and model transformation languages. However, they still need to deal with the syntax and semantics of certain change annotations, and edit operations must be specified in a manual fashion. To this end, generating edit operations automatically from a given meta-model has been proposed (Kehrer et al 2016; Mazanek and Minas 2009; Kehrer et al 2013). However, besides elementary consistency constraints and basic well-formedness rules, meta-models do not convey any domain-specific information on *how* models are edited. Thus, the generation of edit operations from a meta-model is limited to rather primitive operations as a matter of fact. Following the idea of model transformation by-example (MTBE) (Brosch et al 2009; Sun et al 2011; Kappel et al 2012), initial sketches of more complex and domain-specific edit operations can be specified using standard model editors. However, these sketches require manual post-processing to be turned into general specifications, mainly because an initial specification is derived from only a single transformation example. Some MTBE approaches (Kehrer et al 2017; Mokaddem et al 2018) aim at getting rid of this limitation by using a set of transformation examples as input, which are then generalized into a model transformation rule. Still, this is a *supervised* approach, which requires sets of dedicated transformation examples that need to be defined by domain experts in a manual fashion. As discussed by Kehrer et al (2017), a particular challenge is that domain experts need to have, at least, some basic knowledge on the internal processing of the MTBE tool to come up with a reasonable set of examples. Moreover, if only a few examples are used as input for learning, Mokaddem et al (2018) discuss how critical it is to carefully select and design these examples.

To address these limitations of existing approaches, we propose a novel *unsupervised* approach, Ockham, for mining edit operations from existing models in a model repository, which is typically available in large-scale modeling projects (cf. Sect. 2). Ockham is based on an Occam's razor argument, that is, the *useful* edit operations are the ones that *compress* the model repository. In a first step, Ockham discovers frequent change patterns using *frequent subgraph mining* on a labeled graph representation of model differences. It then uses a *compression metric* to filter and rank these patterns. We evaluate Ockham using two controlled experiments with simulated data and one real-world large-scale industrial case study from the railway domain. In the controlled setting, we can show that Ockham is able to discover the edit operations that have been actually applied before by us, even when we apply some perturbation. In the real-world case study, we find that our approach is able to scale to real-world model repositories and to derive edit operations deemed reasonable by practitioners. We evaluated Ockham by comparing the results to randomly generated edit operations in five interviews with practitioners of the product line. We find that the edit operations represent typical edit scenarios and are meaningful to the practitioners. Additionally, we evaluate the practical applicability of the derived edit operations based on a concrete real-world scenario.

In a summary, we make the following contributions:

- We propose an unsupervised approach, called Ockham, that is based on frequent subgraph mining to derive edit operations from model repositories, without requiring any further information.

- We evaluate Ockham empirically based on two controlled simulated experiments and show that the approach is able to discover the applied edit operations.
- We evaluate the approach using an interview with five experienced system engineers and architects from a real-world industrial setting in the railway domain with more than 200 engineers, 300GB of artifacts, and more than 6 years of modeling history. We show that our approach is able to detect meaningful edit operations in this industrial setting and that it scales to real-world repositories.
- We apply the automatically derived edit operations in the concrete use case of condensing large model differences and show their practical impact in this case. We furthermore argue how the edit operations can be used to analyze and simulate model evolution.

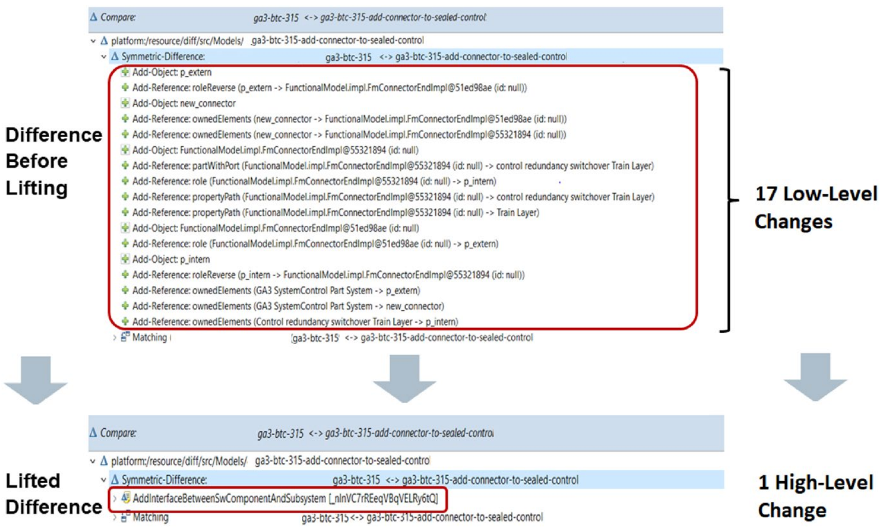
This article is an extension of a conference paper published at ASE'21 (Tinnes et al 2021). In the conference version, we presented and evaluated Ockham, an unsupervised learning approach for mining domain-specific edit operations from model histories. In this article, we include two major additions. First, we evaluate the practical implications of the obtained edit operations using a real-world case study. More specifically, motivated by our aim to facilitate the analysis of model differences, we evaluate the extent to which the edit operations can be used to compress and filter model differences. Second, we discuss key observations that we made in the case study when further analyzing model differences after a semantic lifting using the edit operations mined by Ockham. In particular, we have found that the frequency distribution of the edit operations in the model differences gives rise to a change profile that describes the model difference from a statistical perspective and can be used for classifying the model differences. We discuss applications of these profiles, including the statistical analysis of model differences and the use of the change profiles to simulate model evolution.

2 Motivation: an industrial scenario

Our initial motivation to automatically mine edit operations from model repositories arose from a long-term collaboration with practitioners from a large-scale industrial model-driven software product line in the railway domain. The modeling is done in MagicDraw[1] using SysML, and there is an export to the Eclipse Modeling Framework (EMF), which focuses on the SysML parts required for subsequent MDE activities (e.g., code generation). Modeling tools such as MagicDraw come with support for model versioning. In our setting, the models are versioned in the MagicDraw Teamwork Server. We therefore have access to a large number of models and change scenarios.

Discussing major challenges with the engineers of the product line, we observed that some model changes appear very often together in this repository. For example, when the architect creates an interface between two components, s/

[1] https://docs.nomagic.com/.



```
Δ Compare:    ga3-btc-315 <-> ga3-btc-315-add-connector-to-sealed-control
v Δ platform:/resource/diff/src/Models/ ga3-btc-315-add-connector-to-sealed-control
  v Δ Symmetric-Difference:    ga3-btc-315 <-> ga3-btc-315-add-connector-to-sealed-control
      Add-Object: p_extern
      Add-Reference: roleReverse (p_extern -> FunctionalModel.impl.FmConnectorEndImpl@51ed98ae (id: null))
      Add-Object: new_connector
      Add-Reference: ownedElements (new_connector -> FunctionalModel.impl.FmConnectorEndImpl@51ed98ae (id: null))
      Add-Reference: ownedElements (new_connector -> FunctionalModel.impl.FmConnectorEndImpl@55321894 (id: null))
      Add-Object: FunctionalModel.impl.FmConnectorEndImpl@55321894 (id: null)
      Add-Reference: partWithPort (FunctionalModel.impl.FmConnectorEndImpl@55321894 (id: null) -> control redundancy switchover Train Layer)
      Add-Reference: role (FunctionalModel.impl.FmConnectorEndImpl@55321894 (id: null) -> p_intern)
      Add-Reference: propertyPath (FunctionalModel.impl.FmConnectorEndImpl@55321894 (id: null) -> control redundancy switchover Train Layer)
      Add-Reference: propertyPath (FunctionalModel.impl.FmConnectorEndImpl@55321894 (id: null) -> Train Layer)
      Add-Object: FunctionalModel.impl.FmConnectorEndImpl@51ed98ae (id: null)
      Add-Reference: role (FunctionalModel.impl.FmConnectorEndImpl@51ed98ae (id: null) -> p_extern)
      Add-Object: p_intern
      Add-Reference: roleReverse (p_intern -> FunctionalModel.impl.FmConnectorEndImpl@55321894 (id: null))
      Add-Reference: ownedElements (GA3 SystemControl Part System -> p_extern)
      Add-Reference: ownedElements (GA3 SystemControl Part System -> new_connector)
      Add-Reference: ownedElements (Control redundancy switchover Train Layer -> p_intern)
    > Matching    ga3-btc-315 <-> ga3-btc-315-add-connector-to-sealed-control

Δ Compare:    ga3-btc-315 <-> ga3-btc-315-add-connector-to-sealed-control
v Δ platform:/resource/diff/src/Models/ ga3-btc-315-add-connector-to-sealed-control
  v Δ Symmetric-Difference:    ga3-btc-315 <-> ga3-btc-315-add-connector-to-sealed-control
    > AddInterfaceBetweenSwComponentAndSubsystem [_nlnVC7rREeqVBqVELRy6tQ]
    > Matching    ga3-btc-315 <-> ga3-btc-315-add-connector-to-sealed-control
```

Fig. 1 Semantic lifting can be used to group many fine-grained changes into change sets

he will usually add some Ports to Components and connect them via the ConnectorEnds of a Connector. Expressed in terms of the meta-model, there are 17 changes to add such an interface (see Fig. 1). We are therefore interested to automatically detect these patterns in the model repository. More generally, our approach, OCKHAM, is based on the assumption that it should be possible to derive "meaningful" patterns from the repositories. These patterns could then be used for many applications (Ohrndorf et al 2018; Kögel et al 2016; Taentzer et al 2007; Arendt and Taentzer 2013; Getir et al 2018; Kehrer et al 2012a; ben Fadhel et al 2012; Langer et al 2013; Khelladi et al 2016).

The background is that, in our case study, the models have become huge over time (approx. 1.2 million elements split into 100 submodels) and model differences between different products have become huge (up to 190,000 changes in a single submodel). The analysis of these differences, for example, for quality assurance of the models or domain analysis, has become very tedious and time-consuming. To speed-up the analysis of the model differences, it would be desirable to reduce the "perceived" size of the model difference by grouping fine-grained differences to higher-level, more coarse-grained and more meaningful changes. For this *semantic lifting* of model differences, the approach by Kehrer et al (2011), which uses a set of edit operations as configuration input, can be used but the approach requires the edit operations to be defined already. Based on a set of edit operations this semantic lifting approach will group changes into so called *change sets*.

Large model differences occurring in some of the activities in this setting have actually been our main motivation to investigate how we can derive the required edit operations (semi-)automatically.

We will use the data from this real-world project to evaluate OCKHAM in Sect. 5.

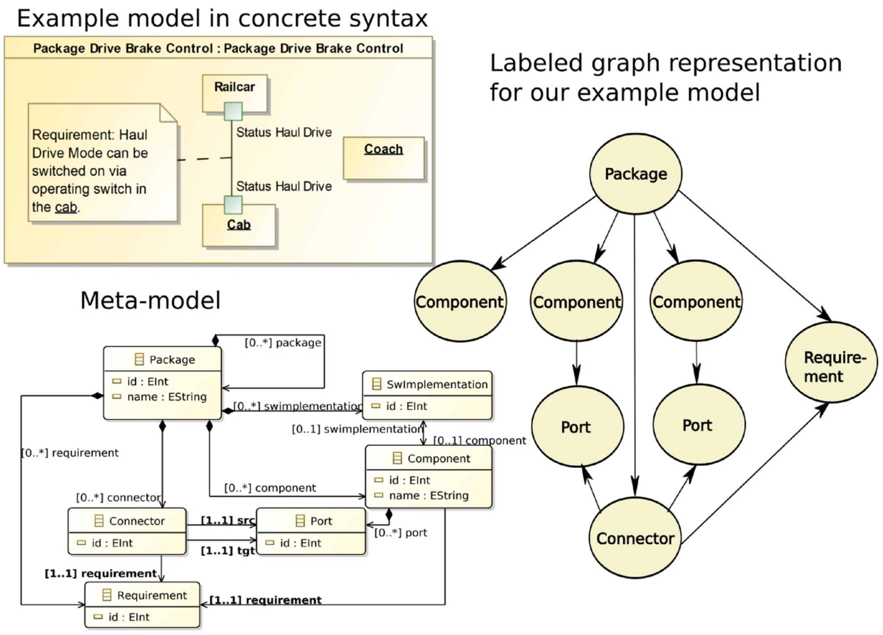


Fig. 2 We consider models as labeled graphs, where labels represent types of nodes and edges defined by a meta-model. For the sake of brevity, types of edges are omitted in the figure

3 Background

In this section, we provide basic definitions that are important to understand our approach presented in Sect. 4.

3.1 Graph theory

As usual in MDE, we assume that a meta-model specifies the abstract syntax and static semantics of a modeling language. Conceptually, we consider a model as a typed graph (aka. abstract syntax graph), in which the types of nodes and edges are drawn from the meta-model. Figure 2 illustrates how a simplified excerpt from an architectural model of our case study from Sect. 2 in concrete syntax is represented in abstract syntax, typed over the given meta-model.

We further assume models to be correctly typed. We abstain from a formal definition of typing using type graphs and type morphisms (Biermann et al 2012), though. Instead, to keep our basic definitions as simple as possible, we work with a variant of labeled graphs in which a fixed label alphabet represents node and edge type definitions of a meta-model. Given a label alphabet L, a *labeled directed graph* G is a tuple (V, E, λ), where V is a finite set of nodes, E is a subset of $V \times V$, called the edge set, and $\lambda : V \cup E \to L$ is the labeling function, which assigns a label to nodes and edges. If we are only interested in the structure of a graph and typing is irrelevant, we will omit the labeling and only refer to the graph as $G = (V, E)$.

Given two graphs $G = (V, E, \lambda)$ and $G' = (V', E', \lambda')$, G' is called a *subgraph* of G, written $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and $\lambda(x) = \lambda'(x)$ for each $x \in V' \cup E'$. A *(weakly) connected component* (component, for short) $C = (V_C, E_C) \subseteq G$ is a subgraph of G in which every two vertices are connected by a *path*, that is, $\forall u, v \in V_C : \exists n \in \mathbb{N}$ s. t. $\{(v, v_1), (v_1, v_2), \dots, (v_n, u)\} \subseteq E_C \cup \tilde{E}_C$, where $\tilde{E}_C$ is the set of all reversed edges, that is, $(u, v) \in E_C$ becomes $(v, u) \in \tilde{E}_C$, and every vertex in G that is connected to a vertex in V_C by a path is an element of V_C.

3.2 Frequent subgraph mining

We will use frequent subgraph mining as the main ingredient for Ockham. We distinguish between *graph-transaction-based frequent subgraph mining* and *single-graph-based frequent subgraph mining*. In particular, we are considering graph-transaction-based frequent subgraph mining, which typically takes a database (i.e., a set) of graphs and a threshold t as input. It then outputs all the subgraphs with, at least, t occurrences in the database. An overview of frequent subgraph mining algorithms can be found in the literature (Jiang et al 2013). A general introduction to graph mining is given by Cook and Holder (2006), who also proposed a compression-based subgraph miner called Subdue (Ketkar et al 2005). Subdue has also been one of our main inspirations for a compression-based approach. Ockham is based on Gaston (Nijssen and Kok 2005), which mines frequent subgraphs by first focusing on frequent paths, then extending to frequent trees, and finally extending the trees to cyclic graphs. Since deciding if a graph G' is a subgraph of another graph G is already $\mathcal{NP}$-complete, there is no polynomial time algorithm for frequent subgraph mining. Usually, if there are large graphs in the database or the threshold is too low, the problem becomes computationally intractable. In these situations, the number of subgraphs typically also becomes intractable. In many scenarios though, one is not interested in a complete list of all frequent subgraphs. In these cases, huge graphs in the database can be filtered out or the threshold can be increased such that the frequent subgraph mining becomes feasible.

3.3 Model transformations and edit operations

The goal of Ockham is to learn domain-specific *edit operations* from model histories. In general, edit operations can be informally understood as editing commands that can be applied to modify a given model. In turn, a difference between two model versions can be described as a (partially) ordered set of applications of edit operations, transforming one model version into the other. Comparing two models can thus be understood as determining the applications of the edit operation applications that transform one model into the other. A major class of edit operations are model refactorings, which induce syntactical changes without changing a models' semantics. Other classes of edit operations include recurring bug fixes and evolutionary changes.

In a classification by Van Deursen et al (2007), edit operations can describe regular evolution, that is, "the modeling language is used to make changes", but they are not meant to describe meta-model evolution, platform evolution, or abstraction evolution. More technically, in Mens et al.'s taxonomy (Mens and Van Gorp 2006), edit operations can be classified as endogenous (i.e., source and target meta-model are equal), in-place (i.e., source and target model are equal) model transformations. For the purpose of this paper, we define an edit operation as an in-place model transformation which represents regular model evolution.

The model transformation tool Henshin (Arendt et al 2010) supports the specification of in-place model transformations in a declarative manner. It is based on graph transformation concepts (Ehrig et al 2004), and it provides a visual language for the definition of transformation rules, which is used, for example, in the last step of Fig. 3. Roughly speaking, transformation rules specify graph patterns that are to be found and created or deleted.

4 Approach

We address the problem of automatically identifying edit operations from a graph mining perspective. As discussed in Sect. 3, we will work with labeled graphs instead of typed graphs. There are some limitations related to this decision, which we discuss in Sect. 6.3.

Ockham consists of the five steps illustrated with a running example in Fig. 3. Our main technical contributions are Step 2 and Step 4. For Step 1, Step 3, and Step 5 we apply existing tooling: SiDiff, Gaston, and Henshin (cf. Sect. 3).

Step 1: Compute Structural Model Differences: To learn a set of edit operations in an *unsupervised* manner, Ockham analyzes model changes that can be extracted from a model's development history. For every pair of successive model versions n and $n+1$ in a given model history, we calculate a *structural model difference* $\Delta(n, n+1)$ to capture these changes. As we do not assume any information (e.g., persistent change logs) to be maintained by a model repository, we use a state-based approach to calculate a structural difference, which proceeds in two steps (Kehrer 2015). First, the corresponding model elements in the model graphs G_n and G_{n+1} are determined using a model matcher (Kolovos et al 2009). In many cases, the model elements will carry identifiers, which can utilized in the matching. Second, the structural changes are derived from these correspondences: All the elements in G_n that do not have a corresponding partner in G_{n+1} are considered to be deleted, whereas, vice versa, all the elements in G_{n+1} that do not have a corresponding partner in G_n are considered to be newly created.

For further processing in subsequent steps, we represent a structural difference $\Delta(n, n+1)$ in a graph-based manner, referred to as *difference graph* (Ohrndorf et al 2018). A difference graph $G_{\Delta(n,n+1)}$ is constructed as a unified graph over G_n and G_{n+1}. That is, corresponding elements being preserved by an evolution step from

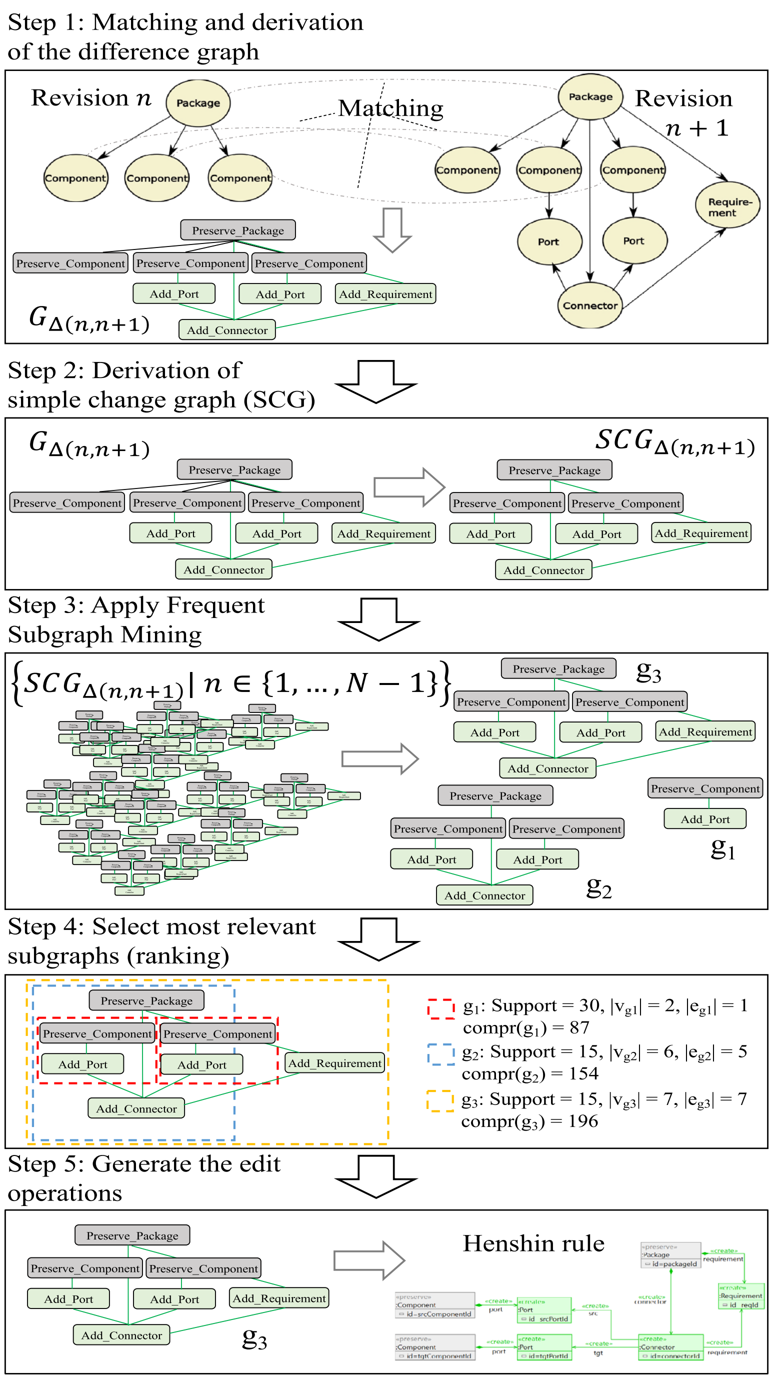

Step 1: Matching and derivation of the difference graph
Revision n
Package
Component
Component
Component
Matching
Package
Revision $n+1$
Component
Component
Component
Require-ment
Port
Port
Connector
Preserve_Package
Preserve_Component
Preserve_Component
Preserve_Component
Add_Port
Add_Port
Add_Requirement
Add_Connector
$G_{\Delta(n,n+1)}$
Step 2: Derivation of simple change graph (SCG)
$G_{\Delta(n,n+1)}$
$SCG_{\Delta(n,n+1)}$
Step 3: Apply Frequent Subgraph Mining
$\{SCG_{\Delta(n,n+1)} \mid n \in \{1,\ldots,N-1\}\}$
g_3
g_1
g_2
Step 4: Select most relevant subgraphs (ranking)
g_1: Support = 30, $|v_{g1}| = 2$, $|e_{g1}| = 1$ compr(g_1) = 87
g_2: Support = 15, $|v_{g2}| = 6$, $|e_{g2}| = 5$ compr(g_2) = 154
g_3: Support = 15, $|v_{g3}| = 7$, $|e_{g3}| = 7$ compr(g_3) = 196
Step 5: Generate the edit operations
g_3
Henshin rule
«preserve» :Package id=packageId
«create» requirement
«create» :Requirement id reqId
«preserve» :Component id=srcComponentId
«create» port
«create» :Port id srcPortId
«create» src
«create» connector
«preserve» :Component id=tgtComponentId
«create» port
«create» :Port id=tgtPortId
«create» tgt
«create» :Connector id=connectorId
«create» requirement

Fig. 3 The 5-step process for mining edit operations with Ockham

version n to $n+1$ appear only once in $G_{\Delta(n,n+1)}$ (indicated by the label prefix "Preserve_"), while all other elements that are unique to model G_n and G_{n+1} are marked as deleted and created, respectively (indicated by the label prefixes "Add_" and "Remove_").

For illustration, assume that the architectural model shown in Fig. 2 is the revised version $n+1$ of a version n by adding the ports along with the connector and its associated requirement. Figure 3 illustrates a matching of the abstract syntax graphs of the model versions n and $n+1$. For the sake of brevity, only correspondences between nodes in G_n and G_{n+1} are shown in the figure, while two edges are corresponding when their source and target nodes are in a correspondence relationship. The derived difference graph $G_{\Delta(n,n+1)}$ is illustrated in Fig. 3. For example, the corresponding nodes of type Component occur only once in $G_{\Delta(n,n+1)}$, and the nodes of type Port are indicated as being created in version $n+1$.

Our implementation is based on the Eclipse Modeling Framework. We use the tool SiDiff (Schmidt and Gloetzner 2008; Kehrer et al 2012b) to compute structural model differences. Our requirements on the model differencing tool are: (1) support for EMF, (2) the option to implement a custom matcher, because modeling tools such as MagicDraw usually provide IDs for every model element, which can be employed by a custom matcher, and (3) an approach to semantically lift model differences based on a set of given edit operations, because we intend to use the semantic lifting approach for the compression of differences in the project mentioned in Sect. 2. Other tools such as EMFCompare could also be used for the computation of model differences and there are no other criteria to favour one over the other. An overview of the different matching techniques is given by Kolovos et al (2009); a survey of model comparison approaches is given by Stephan and Cordy (2013).

Step 2: Derive Simple Change Graphs: Real-world models maintained in a model repository, such as the architectural models in our case study, can get huge. It is certainly fair to say that, compared to a model's overall size, only a small number of model elements is actually subject to change in a typical evolution step. Thus, in the difference graphs obtained in the first step, the majority of difference graph elements represent model elements that are simply preserved. To this end, before we continue with the frequent subgraph mining in Step 3, in Step 2, difference graphs are reduced to *simple change graphs* (SCGs) based on the principle of *locality relaxation*: only changes that are "close" to each other can result from the application of a single edit operation. We discuss the implications of this principle in Sect. 6.3. By "close", we mean that the respective difference graph elements representing a change must be directly connected (i.e., not only through a path of preserved elements). Conversely, this means that changes being represented by elements that are part of different connected components of a simple change graph are independent of each other (i.e., they are assumed to result from different edit operation applications).

More formally, given a difference graph $G_{\Delta(n,n+1)}$, a simple change graph $SCG_{\Delta(n,n+1)} \subseteq G_{\Delta(n,n+1)}$ is derived from $G_{\Delta(n,n+1)}$ in two steps. First, we select all the elements in $G_{\Delta(n,n+1)}$ representing a change (i.e., nodes and edges that are labeled as "Remove_*" and "Add_*", respectively). In general, this selection does not yield a graph, but just a graph fragment $F \subseteq G_{\Delta(n,n+1)}$, which may contain dangling edges. Second, preserved nodes adjacent to dangling edges are also selected to be included in the simple change graph. Formally, the simple change graph is constructed as the boundary graph of F, which is the smallest graph $SCG_{\Delta(n,n+1)} \subseteq G_{\Delta(n,n+1)}$ completing F to a graph (Kehrer 2015). The derivation of a simple change graph from a given difference graph is illustrated in the second step of Fig. 3. In this example, the simple change graph comprises only a single connected component. In a realistic setting, however, a simple change graph typically comprises a larger set of connected components, like the one illustrated in Step 3 of Fig. 3. The simple change graph can also be considered as a blueprint for the edit operation. It contains all the changes performed by the edit operation and the "anchors" it needs in the current model.

We implemented a generator for the simple change graphs in EMF. In our implementation, we loop through all the changes in the symmetric difference and for each change c, we add nodes and edges as below:

AddNode: For an AddNode change, we add a node with label *Add_{TypeName}* to the graph.

RemoveNode: For a RemoveNode change, we add a node with label *Remove_{TypeName}* to the graph.

AddEdge: For a change of type AddEdge, the source object and the target object of the added reference have to be either preserved or also been added to the model m_{i+1}. If the source/target node have not yet been added to the graph, we add a node v_{source}/v_{target} with the labels *Add_{TypeName}* or *Preserve_{TypeName}*, respectively. We then add an edge $e = (v_{source}, v_{target})$ between source and target node to the edge set E of the graph.

RemoveEdge: We add an edge for a change c of type RemoveEdge but the source and target nodes can be either removed or preserved in this case.

AttributeValueChange: An attribute value needs to belong to a preserved object in the model. The corresponding *Preserve_{TypeName}* node is added to the graph if it is not yet there. Then, we add a node with label *Change_{AttributeName}_On_{TypeName}*. The preserved node and the node representing the changed attribute will then be connected by an edge with the same label.

Step 3: Apply Frequent Connected Subgraph Mining: Using the previous two steps for a given model repository that contains a set of model revisions, we can compute a set of Simple Change Graphs. These change graphs contain information about fine-grained, meta-model-based changes between two successive revisions of models. We believe that these fine-grained changes can be grouped to more coarse-grained changes. In the example in Fig. 3, a connector, together with its ports and a requirement, are added. The underlying idea of the approach is that changes that can be *grouped together* should also frequently *appear together* in our model differences. When we apply the first two steps to a model history, we obtain a set of simple change graphs

$$\left\{SCG_{\Delta(n,n+1)} \mid n \in \{1, \dots, N-1\}\right\},$$

where N is the number of revisions in the repository. In this set, we want to identify recurring patterns and therefore find some frequent connected subgraphs. A small support threshold might lead to a huge number of frequent subgraphs. For example a support threshold of one would yield every subgraph in the set of connected components. This does not only cause large computational effort, but also makes it difficult to find relevant subgraphs. As it would be infeasible to recompute the threshold manually for every dataset, we pre-compute it by running an approximate frequent *subtree miner* for different thresholds up to some fixed size of frequent subtrees. We fix the range of frequent trees and adjust the threshold accordingly. More precisely, we perform a binary search for the threshold until the number of subtrees lies in a predefined range. We use this threshold for the exact subgraph mining. This effectively replaces the support threshold hyper-parameter by a hyper-parameter ST_{max} which satisfies

$$ST(level = 8, miner) < ST_{max}$$

where $ST(level = 8, miner)$ denotes the number of subtrees found up to level (number of nodes) 8 with the frequent subtree miner used. We therefore do not need to (manually) find a good support threshold for the datasets and, in many cases, we can be sure that the miner is able to terminate in a reasonable amount of time.

Alternatively, a relative threshold could be used, but we found in a pilot study that our pre-computation works better in terms of average precision. We discuss the effect of the support threshold further in Sect. 6.

We run the frequent subgraph miner for the threshold found via the approximate tree miner. Step 3 of Fig. 3 shows this for our running example. We start with a set of connected components, and the graph miner returns a set of frequent subgraphs, namely $\{g_1, g_2, g_3\}$ with $g_1 \subset g_2 \subset g_3$. We use the Gaston (Nijssen and Kok 2005) graph miner, since it performed best (in terms of runtime) among the miners that we experimented with (gSpan, Gaston, and DIMSpan) in a pilot study. In our pilot study, we ran the miners on a small selection of our datasets and experimented with the parameters of the miners. For many datasets, gSpan and DIMSpan did not terminate at all (we canceled the execution after 48 h). Gaston (with embedding lists) was able to terminate in less then 10 s on most of our datasets but consumes a lot of memory, typically between 10GB–25GB, which was not a problem for our 32GB machine in the pilot study. To rule out any effects due to approximate mining, we considered only exact miners. Therefore, we also could not use Subdue (Ketkar et al 2005), which directly tries to optimize compression. Furthermore, Subdue was not able to discover both edit operations in the second experiment (see Sect. 5), without iterative mining and allowing for overlaps. Enabling these two options, Subdue did not terminate on more than 75% of the pilot study datasets. For frequent subtree mining, we use Hops (Welke et al 2020) because it provides low error rates and good runtime guarantees.

Step 4: Select the most Relevant Subgraphs: Motivated by the *minimum description length principle*, which has been successfully applied to many different kinds of data (Grünwald and Grunwald 2007), the most relevant patterns should not be the most frequent ones but the ones that give us a maximum compression for our original data (Djoko 1994). That is, we want to express the given SCGs by a set of subgraphs such that the description length for the subgraphs together with the length of the description of the SCGs in terms of the subgraphs becomes minimal. This reasoning can be illustrated by looking at the corner cases: (1) A single change has a large frequency but is typically not interesting. (2) The entire model difference is large in terms of changes but has a frequency of only one and is typically also not an interesting edit operation. "Typical edit operations" are therefore somewhere in the middle. We will use our experiments in Sect. 5 to validate whether this assumption holds. We define the compression value by

$$\mathrm{compr}(g) = \big(\mathrm{supp}(g) - 1\big) \cdot \big(\mid V_g \mid + \mid E_g \mid \big) ,$$

where $\mathrm{supp}(g)$ is the support of g in our set of input graphs (i.e., the number of components in which the subgraph is contained). The "−1" in the definition of the compression value comes from the intuition that we need to store the definition of the subgraph, to decompress the data again. The goal of this step is to detect the subgraphs from the previous step with a high compression value. Subgraphs are organized in a *subgraph lattice*, where each graph has pointers to its direct subgraphs. Most of the subgraph miners already compute a subgraph lattice, so we do not need a subgraph isomorphism test here. Due to the downward closure property of the support, all subgraphs of a given (sub-)graph have, at least, the same frequency (in transaction-based graph mining). When sorting the output, we need to take this into account, since we are only interested in the largest possible subgraphs for some frequency. These largest possible subgraphs for some frequency are called closed subgraphs and typically the number of closed subgraphs is much smaller than the number of all frequent subgraphs (Yan and Han 2003). Therefore, we prune the subgraph lattice. The resulting list of recommendations is then sorted according to the compression value. Other outputs are conceivable, but in terms of evaluation, a sorted list is a typical choice for a recommender system (Schröder et al 2011).

More technically, let SG be the set of subgraphs obtained from Step 3, we then remove all the graphs in the set

$$\begin{aligned} SG^- = \big\{ g \in SG \mid \exists \tilde{g} \in SG, \text{ with } g \subseteq \tilde{g} \\ \wedge \mathrm{supp}(g) = \mathrm{supp}(\tilde{g}) \wedge \mathrm{compr}(g) \le \mathrm{compr}(\tilde{g}) \big\}. \end{aligned}$$

Our list of recommendations is then $SG \setminus SG^-$, sorted according to the compression metric.

For our running example in Step 4 of Fig. 3, assume that the largest subgraph g_3 occurs 15 times (without overlaps). Even though the smaller subgraph g_1 occurs twice as often, we find that g_3 provides the best compression value and is therefore ranked first. Subgraph g_2 will be pruned, since it has the same support as its supergraph g_3, but a lower compression value.

We implement the compression computation and pruning using the *NetworkX*[2] Python library. The algorithm outputs a compression-ranked and pruned list of frequent subgraphs.

[2] https://networkx.org/.

Step 5: Generate Edit Operations: As a result of Step 4, we have an ordered list of “relevant” subgraphs of the simple change graphs. We need to transform these subgraphs into model transformations that specify the mined edit operations. As illustrated in Step 5 of Fig. 3, the subgraphs can be transformed to Henshin transformation rules in a straightforward manner. The added elements are added to the right hand side graph of the Henshin transformation rule, the deleted elements are added to the left hand side graph of the Henshin transformation rule, and preserved elements are added to both graphs and a corresponding matching is added to the rule. At this point, we can also use the information from the meta-model to complete the edit operations. For example, when a created element requires another element or attribute, which is not yet present, it can be added. This is done similarly to the derivation of consistency preserving edit operations by Kehrer et al (2016). A domain expert can add some (non-)application conditions to the rules or add further parameters for the rules. We use Henshin because it is used for the semantic lifting approach in our case study from Sec. 2. In principle, any transformation language that allows us to express endogenous, in-place model transformations could be used. A survey of model transformation tools is given by Kahani et al (2019).

5 Evaluation

In this section, we will evaluate our approach in two controlled experiments and one real-world industry case study in the railway domain.

5.1 Research questions

We evaluate Ockham w.r.t. the following research questions:

- *RQ* 1: *Is* Ockham *able to identify edit operations that have actually been applied in model repositories?* If we apply some operations to models, Ockham should be able to discover these from the data. Furthermore, when different edit operations are applied and overlap, it should still be possible to discover them.
- *RQ* 2: *Is* Ockham *able to find typical edit operations or editing scenarios in a real-world setting?* Compared to the first research question, Ockham should also be able to find typical scenarios in practice, in scenarios where we do not know which operations have been actually applied to the data. Furthermore, it should be possible to derive these edit operations in a real-world setting with large models and complex meta-models.
- *RQ* 3: *What are the main drivers for* Ockham *to succeed or fail?* We want to identify the characteristics of the input data or parameters having a major influence on Ockham.
- *RQ* 4: *What are the main parameters for the performance of the frequent subgraph mining?* Frequent subgraph mining has a very high computational com-

plexity for general cyclic graphs. We want to identify the characteristics of the data in our setting that influence the mining time.

- *RQ* 5: *What is the practical impact of the edit operations discovered by* Ockham *in the context of the case study introduced in Section* 2? One of our main motivations is to compress differences in our case study of Sect. 2. We evaluate which degree of compression our mined operations provide in this scenario.

For RQ 1, we want to rediscover the edit operations from our ground truth, whereas in RQ 2, the discovered operations could also be some changes that are not applied in "only one step" but appear to be typical for a domain expert. We refer to both kinds of change patterns as "meaningful" edit operation.

5.2 Experiment setup

We conduct four experiments to evaluate our approach. In the first two experiments, we run the algorithm on synthetic model repositories. We know the "relevant edit operations" in these repositories, since we define them, and apply them to sample models. We can therefore use these experiments to answer RQ 1. Furthermore, since we are able to control many properties of our input data for these simulated repositories, we can also use them to answer RQ 3 and RQ 4. In the third experiment, we apply Ockham to the dataset from our case study presented in Sect. 2 to answer RQ 2. The first two experiments help us to find the model properties and the parameters the approach is sensible to. Their purpose is to increase the *internal validity* of our evaluation. To increase *external validity*, we apply Ockham in a real-world setting as well. None of these experiments alone achieves sufficient internal and external validity (Siegmund et al 2015), but the combination of all experiments is suitable to assess whether Ockham can discover relevant edit operations. In the fourth experiment, we use the edit operations from the third experiment to semantically lift the low-level differences using the approach of Kehrer et al (2011). We then compute the compression ratio on a sample of model differences to answer RQ 5.

We run the experiments on an Intel Core$^{\text{TM}}$ i7-5820K CPU @ 3.30GHz× 12 and 31.3 GiB RAM. For the synthetic repositories, we use 3 cores per dataset.

Experiment 1: As a first experiment, we simulate the application of edit operations on a simple component model. The meta-model is shown in Fig. 2.

For this experiment, we only apply one kind of edit operation (the one from our running example in Fig. 3) to a random model instance. The Henshin rule specifying the operation consists of a graph pattern comprising 7 nodes and 7 edges. We create the model differences as follows: We start with an instance m_0 of the simple component meta-model with 87 Packages, 85 Components, 85 SwImplementations, 172 Ports, 86 Connectors, and 171 Requirements. Then, the edit operation is randomly applied e times to the model obtaining a new model revision m_1. This procedure is applied iteratively d times to obtain the model history $m_0 \rightarrow m_1 \rightarrow \ldots m_{d-1} \rightarrow m_d$. Each evolution step $m_i \rightarrow m_{i+1}$ yields a difference $\Delta(m_i, m_{i+1})$.

Since we can not ensure completeness of Ockham (i.e., it might not discover all edit operations in a real-world setting), we also have to investigate how sensible the approach is to undiscovered edit operations. Therefore, to each application of the edit operation, we apply a random perturbation. More concretely, a perturbation is another edit operation that we apply with a certain probability p. This perturbation is applied such that it overlaps with the application of the main edit operation. We use the tool Henshin (Biermann et al 2012) to apply model transformations to one model revision. We then build the difference of two successive models as outlined in Sect. 4. In our experiment, we control the following parameters for the generated data.

- d: The number of differences in each simulated model repository. For this experiment, $d \in \{10, 20\}$.
- e: The number of edit operations to be applied per model revision in the repository, that is, how often the edit operation will be applied to the model. For this experiment, $e \in \{1, \dots, 100\}$.
- p: The probability that the operation will be perturbed. For this experiment, we use $p \in \{0.1, 0.2, \dots, 1.0\}$.

This gives us 2000 (= $2 \times 100 \times 10$) datasets for this experiment. A characteristics of our datasets is that, increasing e, the probability of changes to overlap increases, as well. Eventually, adding more changes even decreases the number of components in the SCG while increasing the average size of the components.

Ockham suggests a ranking of the top k subgraphs (which eventually yield the learned edit operations). In the ranked suggestions of the algorithm, we then look for the position of the "relevant edit operation" by using a graph isomorphism test. To evaluate the ranking, we use the "mean average precision at k" (MAP@k), which is commonly used as an accuracy metric for recommender systems (Schröder et al 2011):

$$\text{MAP@k} := \frac{1}{|D|} \sum_{\text{D}} \text{AP@k},$$

where D is the family of all datasets (one dataset represents one repository) and AP@k is defined by

$$\text{AP@k} := \frac{\sum_{i=1}^{k} \text{P}(i) \cdot \text{rel}(i)}{|\text{total set of relevant subgraphs}|},$$

where $\text{P}(i)$ is the precision at i, and $\text{rel}(i)$ indicates if the graph at rank i is relevant.

For this experiment, the number of relevant edit operations (or subgraphs to be more precise) is always one. Therefore, we are interested in the rank of the correct edit operation. Except for the case that the relevant edit operation does not show up at all, MAP@∞ gives us the mean reciprocal rank and therefore serves as a good metric for that purpose.

For comparison only, we also compute the MAP@k scores for the rank of the correct edit operations according to the frequency of the subgraphs. Furthermore, we investigate how the performance of subgraph mining depends on other parameters of Ockham. We are also interested in how average precision (AP), that is, AP@∞, depends on the characteristics of the datasets. Note that for the first two experiments, we do not execute the last canonical step of our approach (i.e., deriving the edit operation from a SCG), but we directly evaluate the resulting subgraph from Step 4 against the simple change graph corresponding to the edit operation.

To evaluate the performance of the frequent subgraph miner on our datasets, we fixed the relative threshold (i.e., the support threshold divided by the number of components in the graph database) to 0.4. We re-run the algorithm for this fixed relative support threshold and $p \leq 0.4$.

Experiment 2: In contrast to the first experiment, we want to identify in the second experiment more than one edit operation in a model repository. We therefore extent the first experiment by adding another edit operation, applying each of the operations with the same probability. To test whether Ockham also detects edit operations with smaller compression than the dominant (in terms of compression) edit operation, we choose a smaller second operation. The Henshin rule graph pattern for the second operation comprises 4 nodes and 5 edges. It corresponds to adding a new Component with its SwImplementation and a Requirement to a Package.

Since the simulation of model revisions consumes a lot of compute resources, we fixed $d = 10$ and considered only $e <= 80$ for this experiment. The rest of the experiment is analogous to the first experiment.

Experiment 3: The power of the simulation to mimic a real-world model evolution is limited. Especially, the assumption of random and independent applications of edit operations is questionable. Therefore, for the third experiment, we use a real-world model repository from the railway software development domain (see Sect. 2). For this repository, we do not know the operations that have actually been applied. We therefore compare the mined edit operations with edit operations randomly generated from the meta-model, and want to show that the mined edit operations are significantly more “meaningful” than the random ones.

The models in this case are SysML models in MagicDraw but there is an EMF export of the SysML models, which we can use to apply our toolchain.

For this experiment, we mined 546 pairwise differences, with 4109 changes, on average, which also contain changed attribute values (one reason for that many changes is that the engineering language has changed from German to English). The typical model size in terms of their abstract syntax graphs is 12081 nodes; on average, 50 out of 83 meta-model classes are used as node types.

To evaluate the quality of our recommendations, we conducted a semi-structured interview with five domain experts of our industry partner: 2 system engineers working with one of the models, 1 system engineer working cross-cutting, 1 chief system architect responsible for the product line approach and the head of the tool development team. We presented them 25 of our mined edit operations together with 25 edit operations that were randomly generated out of the meta-model. We selected the mined edit operations that were top-ranked according to compression metric. We omitted repeated edit operations that result from multi-object structures (e.g., if the recommendations included edit operations for adding two activity diagram swimlanes, adding three swimlanes, etc., we only included one of them). We used the compression-based ranking since it is arguably superior to the purely frequency-based ranking, as we have seen empirically in the first two experiments. Furthermore, there is also more evidence for this choice in the literature (Ketkar et al 2005; Bariatti et al 2020). The edit operations were presented in the visual transformation language of Henshin, which we introduced to our participants before. On a 5-point Likert scale, we asked whether the edit operation represents a typical edit scenario (5), is rather typical (4), can make sense but is not typical (3), is unlikely to exist (2), and does not make sense at all (1). We compare the distributions of the Likert score for the population of random edit operations and mined edit operations to determine whether the mined operations are typical or meaningful.

In addition, we discussed the mined edit operations with the engineers that have not been considered to be typical.

Experiment 4: Higher-level edit operations typically comprise several more fine-grained edit steps. Kehrer et al (2011) presented an approach that allows to represent a model difference in terms of previously defined edit operations. Changes in the model difference that belong to the application of one edit operation are grouped together in a *lifted model difference*. A group of changes that belong to the application of one edit operation is called a *change set*. Using this lifting approach, we can evaluate to what extend the edit operations mined by Ockham can compress model differences. For this, in the fourth experiment, we use the 25 mined edit operations from the third experiment and five additional edit operations discovered by Ockham for a semantic lifting of a sample of 40 model differences from our industrial scenario from Sect. 2. More concretely, we randomly select 40 submodels in their current revision and compute the difference between this revision and 25 revisions earlier of the same submodel. If no 25 revisions are available, we compute the difference to the earliest possible revision. Of course, our 30 edit operations are not yet complete, that is, some changes will be missed. We will therefore not only report the total compression ratio but also the relative compression ratio (i.e., the compression ratio with respect to the changes that are covered by the 30 edit operations). We define the compression ratio as follows:

$$\mathrm{c} := \frac{|\text{changes before lifting}|}{|\text{change sets}| + |\text{ungrouped changes}|},$$

where by *ungrouped changes* we refer to changes that are not included in any of the change sets after the semantic lifting. Similarly, the relative compression ratio is defined as:

$$\mathrm{c}_{\mathrm{rel}} := \frac{|\text{changes before lifting}| - |\text{ungrouped changes}|}{|\text{change sets}|}$$

With the help of the domain experts we furthermore divided the set of 30 edit operations into two subsets. One subset contains refactorings like renaming operations or move operations, while the other subset contains edit operations which represent important functional evolution. We will refer to the second subset as *critical edit*

Table 1 The MAP@k scores for results of Experiment 1

	MAP@1	MAP@5	MAP@10	MAP@∞
Compression	0.967	0.974	0.975	0.975
Frequency	0.016	0.353	0.368	0.368

Table 2 The MAP@k scores for Experiment 2

	MAP@2	MAP@5	MAP@10	MAP@∞
Compression	0.955	0.969	0.969	0.969
Frequency	0.013	0.127	0.152	0.190

Table 3 Spearman correlations for Experiment 1

	p	Mining time	e	d	Mean #Nodes per comp
AP	– 0.07*	– 0.24**	– 0.23**	0.12**	– 0.21**
AP (for e > 80)	– 0.14*	– 0.19**	– 0.19**	0.25**	– 0.03
Mining time	0.12**	–	0.89**	0.26**	0.83**

**: p < .001, *: p < 0.01

operations. 22 edit operations have been classified as critical and 8 as refactorings. We will also report the compression ratio for these critical operations separately. The reason for this separate reporting is that for some of the tasks related to model differences, refactoring is less relevant and can be ignored, while the critical edit operations are crucial for the analysis. An example for such a task is the analysis of the functional changes of a product between two successive software qualifications. It is important to emphasize here that the discovery of refactoring edit operations is still important and a prerequisite for this filtering.

5.3 Results

Experiment 1: In Table 1, we list the MAP@k scores for all datasets in the experiment. Table 3 shows the Spearman correlation of the independent and dependent variables. If we look only on datasets with a large number of applied edit operations, $e > 80$, the Spearman correlation for average precision vs. d and average precision vs. p becomes 0.25 (instead of 0.12) and −0.14 (instead of −0.07), respectively. The mean time for running Gaston on our datasets was 1.17 s per dataset.

Experiment 2: In Table 2 we give the MAP@k scores for this experiment. Table 4 shows the correlation matrix for the second experiment. The mean time for running GASTON on our datasets was 1.02 s per dataset.

Table 4 The Spearman correlation matrix for Experiment 2

	p	Size at threshold	Mining time	e	Mean #Nodes per comp
AP	– 0.20**	0.07	– 0.02	0.09*	0.02
p	–	0.23**	0.16**	0	0.30**
Size at threshold	–	–	0.54**	0.57**	0.65**
Mining time	–	–	–	0.75**	0.75**
e	–	–	–	–	0.92**

**: $p < .001$, *: $p < 0.01$

Table 5 Statistics for the Likert values for Experiment 3

Participant	Mean mined	Mean random	*p*- value (*t*-test)	*p*- value (Wilcoxon)	*p*- value (bootstrap)
P1	3.20	1.68	$11.8 \cdot 10^{-5}$	$29.0 \cdot 10^{-5}$	$11.0 \cdot 10^{-4}$
P2	4.04	2.76	$16.6 \cdot 10^{-4}$	$6.43 \cdot 10^{-3}$	$3.60 \cdot 10^{-3}$
P3	4.32	2.60	$9.30 \cdot 10^{-6}$	$5.87 \cdot 10^{-5}$	$2.00 \cdot 10^{-4}$
P4	4.32	1.08	$2.67 \cdot 10^{-15}$	$3.51 \cdot 10^{-10}$	$< 2.20 \cdot 10^{-16}$
P5	4.48	1.60	$1.17 \cdot 10^{-11}$	$1.15 \cdot 10^{-7}$	$9.00 \cdot 10^{-4}$
Total	4.072	1.944	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$

Experiment 3: Table 5 shows the results for the Likert values for the mined and random edit operations for the five participants of our study. Furthermore, we conduct a t-test (with and without bootstrap) and a Wilcoxon signed-rank test, to test if the mined edit operations more likely present typical edit scenarios than the random ones. The p-values are reported in Table 5.

Null hypothesis $\mathbf{H_0}$: *The mined edit operations do not present a more typical edit scenario than random edit operations on average.*

We set the significance level to $\alpha = 0.01$. We can see that, for all participants, the mean Likert score for the mined operations is significantly (for all statistical tests performed) higher than the mean for the random operations. Furthermore, the 95th percentile confidence interval for the difference of the means between the two groups (i.e., mined edit operations Likert values and random edit operations Likert values) is [1.80, 2.46]. This interval does not contain the null value. We can therefore reject the null hypothesis.

Note that we also correlated the frequency of the edit operations in our dataset with the average Likert value from the interviews. We found a small but insignificant correlation (Pearson correlation of 0.33, $p = 0.12$).

After their rating, when we confronted the engineers with the true results, they stated that the edit operations obtained by Ockham represent typical edit scenarios. According to one of the engineers, some of the edit operations "can be slightly

Table 6 Statistics for the semantic lifting

	Total changes	Changes lifted	Unlifted changes	Relative compr.	Total compr.
Total	1, 302, 591	1, 006, 136	907, 361	4.00	1.29
$\text{Total}_{\text{crit}}$	1, 302, 591	1, 040, 814	987, 194	5.88	1.25
Average	32, 564.78	25, 153.40	22, 684.03	3.74	1.41
Std. Dev.	27, 321.05	21, 248.99	19, 425.70	1.52	0.28
Min	16.00	10.00	0.00	1.91	1.10
Max	73, 70.006	56, 032.00	51, 500.00	10.65	2.22

extended" (see also Sect. 6). Some of the edit operations found by Ockham, but not recognized by the participants, were identified "to be a one-off refactoring that has been performed some time ago".

In this real-world repository, we found some operations that are typical to the modeling language SysML, for example, one which is similar to the simplified operation in Fig. 3. We also found more interesting operations, for example, the addition of ports with domain-specific port properties. Furthermore, we were able to detect some rather trivial changes. For example, we can see that typically more than just one swimlane is added to an activity, if any. We also found simple refactorings, such as renaming a package (which also leads to changing the fully qualified name of all contained elements) or also refactorings that correspond to changed conventions, for example, activities were owned by so called System Use Cases before but have been moved into Packages.

Experiment 4: When performing the semantic lifting with the 30 selected edit operations on the 40 model differences, we first observe that we matched only 30% of the total changes in our model differences. Furthermore, with respect to our classification of the edit operations in noncritical and critical edit operations, we observe that noncritical edit operations occur more than twice as often as critical edit operations. We had 13, 870 occurrences, on average, for a noncritical edit operation and 6, 027 occurrences for a critical edit operation.

In Table 6, we report the number of changes before the semantic lifting, the number of changes after lifting (i.e., changes not covered by any change set plus the number of change sets), the number of "unlifted changes" (i.e., changes not covered by any change set), the relative compression c_{rel}, and the total compression c. Our sample of 40 model differences consists of $\sim$ 1.3 million changes. With the exception of one edit operation, we found all edit operations in our samples. On average an edit operation has 8, 189 occurrences in the dataset. We report these values for all 40 model differences together (referred to as "Total" in Table 6), for the average over the 40 model differences (referred to as "Average" in Table 6), their standard deviation (referred to as "Std. Dev." in Table 6), the minimum, and the maximum over the 40 model differences. Furthermore, we also report the compression ratios for the case where we restricted the semantic lifting to the set of critical edit operations.

We observe a relative compression ratio c_{rel} (see definition above) of 4.00 over all 40 model differences. In contrast, the total compression ratio c is 1.29. When

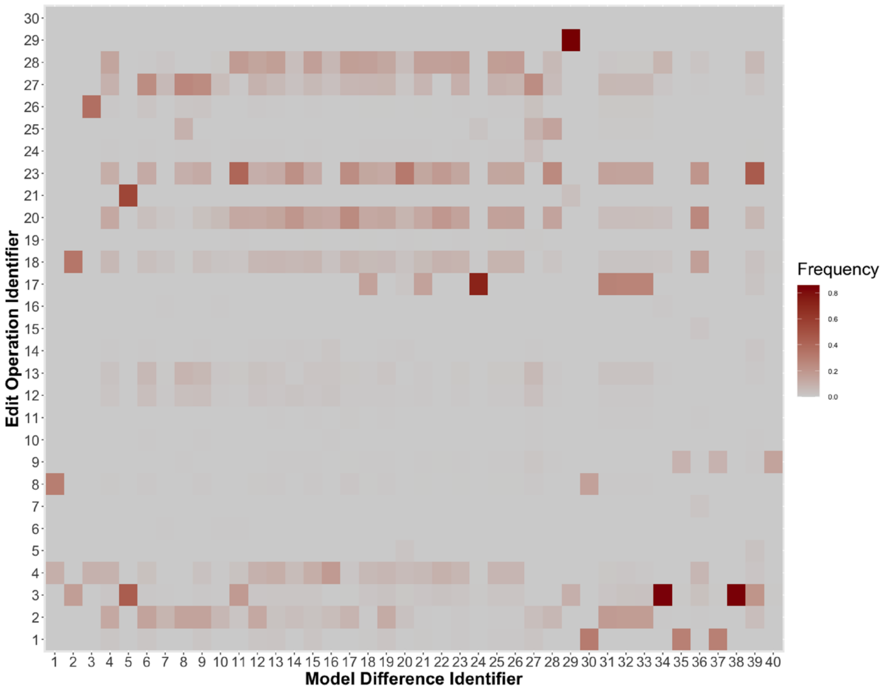


Fig. 4 This plot shows for every model difference the distribution of the edit operations. That is, every column is a kind of change profile for the corresponding model difference. The darker the color, the higher the (relative) frequency of the edit operation in this model difference

restricting the set of edit operations in the semantic lifting to the set of critical edit operations, we achieve a relative compression ratio c_{rel} of 5.88. Since we are considering a smaller set of edit operations for the lifting in this case, the total compression ratio for our critical edit operations is smaller.

Another observation that can be made from the semantic lifting is that the distribution of the occurrences of edit operations over the model differences are rather heterogeneous (see Fig. 4), even though for some of the model differences their distribution highly correlates. We elaborate on this observation in Sect. 6.2.

6 Discussion

6.1 Research questions

RQ 1: *Is* Ockham *able to identify relevant edit operations in model repositories?* We can answer this question with a "yes". Experiment 1 and 2 show high MAP scores. Only for a large number of applied operations and a large size of the input graphs, Ockham fails in finding the applied edit operations. We can see that our compression-based approach clearly outperforms the frequency-based approach used as a baseline.

Table 7 Main drivers for Ockham to fail in experiment 1

	p	Mean #Nodes per component	Size at threshold	Mining time
Overall mean	0.55	57.6	8.20	1.26
Mean for undetected operation	0.79	109.0	10.03	2.55

RQ 2: *Is* **Ockham** *able to find typical edit operations or editing scenarios in a real-world setting?*

The edit operations found by Ockham obtained significantly higher (mean/median) Likert scores than the random edit operations. Furthermore we observe a mean Likert score of almost 4.1. From this we can conclude that, compared to random ones, our mined edit operations can be considered as typical edit scenarios, on average. When looking at the mined edit operations it becomes clear, that Ockham is able to implicitly identify constraints, which where not made explicit in the meta-model. Also, except for one edit operation, the mined edit operations appear in the sample dataset from experiment 4. The edit operations recommended by Ockham are correct in most cases, and incomplete edit operations can be adjusted manually. We cannot state yet that the approach is also *complete* (i.e., is able to find all relevant edit scenarios), though.

RQ 3: *What are the main drivers for* **Ockham** *to succeed or fail?*

From Table 3, we observe that increasing the number of edit operations has a negative effect on the average precision. Increasing the perturbation has a slightly negative effect, which becomes stronger for a high number of applied edit operations and therefore when huge connected components start to form. The number of differences d (i.e., having more examples) has a positive effect on the rank, which is rather intuitive. For the second experiment, from Table 4, we can observe a strong dependency of the average precision on the perturbation parameter, which is, stronger than for the first experiment. On the other hand, the correlation to the number of applied edit operations is even positive.

To analyze the main drivers further, we take a deeper look into the results. We have to distinguish between the two cases that (1) the correct edit operation is not detected at all and (2) the correct edit operation has a low rank.

Edit operation has not been detected: For the second experiment, in 22 out of 800 examples, Ockham was not able to detect both edit operations. In 10 of these cases the threshold has been set too high. To mitigate this problem, in a real-world setting, the threshold parameters could be manually adjusted until the results are more plausible. In the automatic approach, further metrics have to be integrated.

Other factors that cause finding the correct edit operations to fail are the perturbation, average size of component, and the size at threshold, as can be seen from Table 7. Given a support threshold t, the *size at threshold* is the number of nodes of the t-largest component. The intuition behind this metric is the following: For the frequent subgraph miner, in order to prune the search space, a subgraph is only

Table 8 Possible drivers (number of differences (d), number of applied edit operations (e), perturbation (p), mean number of nodes per component, size at threshold) for a low rank (≥ 5)

d	e	p	Mean #Nodes per component	Size at threshold	Average precision	Rank
10	92	0.3	142.2	13	0.13	8
10	67	0.4	91.0	16	0.14	7
10	78	0.8	87.3	14	0.14	7
10	98	0.8	127.7	14	0.067	15
20	81	0.1	227.0	16	0.13	8
20	99	0.1	272.2	19	0.010	99
20	100	0.1	272.7	17	0.013	78

allowed to appear in, at most, $t-1$ components. Therefore, the subgraph miner needs to search for a subgraph, at least, in one component with size greater than the *size at threshold*. Usually, the component size plays a major role in the complexity of the subgraph mining. When the t-largest component is small, we could always use this component (or smaller ones) to guide the search through the search space and therefore we will not have a large search space. So, a large size of the component at threshold could be an indicator for a complicated dataset.

Looked deeper into the results of the datasets from the first experiment, for which the correct subgraph has not been identified, we can see that, for some of these subgraphs, there is a supergraph in our recommendations that is top-ranked. Usually this supergraph contains one or two additional nodes. Since we have a rather small meta-model, and we only use four other edit operations for the perturbation, it can happen rarely that these larger graphs occur with the same frequency as the actual subgraph. The correct subgraphs are then pruned away.

Edit operation has a low rank: First, note that we observe a low rank (rank ≥ 5) only very rarely. For the first experiment, it happened in 7 out of 2000 datasets, while for the second experiment, it did not happen at all. In Table 8, we list the corresponding datasets and the values for drivers of a low rank.

One interesting observation is that, for some of the datasets with low-ranked correct subgraph, we can see that the correct graph appears very early in the subgraph lattice, for example, first child of the best compressing subgraph but rank 99 in the output, or first child of the second best subgraph but rank 15 in the output. This suggests that this is more a presentation issue, which is due to the fact that we have to select a linear order of all subgraph candidates for the experiment.

In Experiment 3, we only found two mined edit operations that received an average Likert score below 3 from the five practitioners in the interviews. The first one was a refactoring that was actually performed but that targeted only a minority of all models. Only two of the participants where aware of this refactoring, and one of them did not directly recognize it due to the abstract presentation of the refactoring. The other edit operation that was also not considered as a typical edit scenario was adding a kind of document to another document. This edit operation was even considered as illegal by 3 out of the 5 participants. The reason for this is the internal modeling of the relationship between the documents, which the participants were not aware of. So, it can also be attributed to the presentation of the results in terms of Henshin rules, which require an understanding of the underlying modeling language's meta-model.

For four of the edit operations of Experiment 3, some of the participants mentioned that the edit operation can be extended slightly. We took a closer look at why Ockham was not able to detect the extended edit operation, and it turned out that it was due to our simplifications of locality relaxation and also due to the missing type hierarchies in our graphs. For example, in one edit operation, one could see that the fully qualified name (name + location in the containment hierarchy) of some nodes has been changed, but the actual change causing this name change was not visible, because it was a renaming of a package a few levels higher in the containment hierarchy that was not directly linked to our change. Another example was a "cut off" referenced element in an edit operation. The reason why this has been cut off was that the element appeared as different sub-classes in the model differences and each single change alone was not frequent.

To summarize: The main drivers for Ockham to fail are a large average size of components and the size at threshold. The average size is related to the number of edit operations applied per model difference. In a practical scenario, huge differences can be excluded when running edit operation detection. The size of the component at threshold can be reduced by increasing the support threshold parameters of the frequent subgraph mining. With higher threshold, we increase the risk of missing some less frequent edit operations, but the reliability for detecting the correct (more frequent) operations is increased. Having more examples improves the results of Ockham.

RQ 4: *What are the main parameters for the performance of the frequent subgraph mining?* From Table 3, we can observe a strong Spearman correlation of the mining time with the number of applied edit operations e (0.89) and implicitly also the average number of nodes per component (0.83). If we only look at edit operations with rank > 1, we observe a strong negative correlation of -0.51 with the average precision (not shown in Table 3). This actually means that large mining times usually come with a bad ranking. The same effect can be observed for Experiment 2 (Table 4). We can also see, that the mining time correlates with the size at threshold.

RQ 5: *What is the practical impact of the edit operations discovered by* Ockham *in the context of the case study in Section 2*? In experiment 4, we lifted the changes using the edit operations discovered by Ockham and applied the classification into refactorings and critical edit operations. From this experiment we can see, first of all, that our edit operations also occur in practice. There is only one edit operation that did not occur at all in our example models. The reason is that this edit operation corresponds to a one-time refactoring that was done some time ago, but was not included in our randomly selected sample. Furthermore, the edit operations mined by Ockham can be used to compress model differences significantly. Since they are also meaningful to domain experts (see RQ 2), they can be used to summarize model differences. Rather "small" changes, for example, adding a package ($\sim$ 23% of the change sets) or renaming a property ($\sim$ 10% of the change sets), occur with a much higher frequency than more complex edit operations such as adding an external port with multiplicities ($\sim 1.3\%$ of the change sets). In general, from the results of experiment 4, we can see that critical edit operations are more rare compared to refactoring edit operations. On the other hand, they provide a higher compression of the model differences. The classification of the edit operations could also be used to filter out irrelevant changes in the analysis of model differences. Nevertheless, the classification is most likely task-dependent and we cannot be sure that this observation also holds for other classifications.

Furthermore, the the distribution of edit operations in a model difference can tell us something about the nature of the evolution step. For example, we can determine “hot spots”, where a lot of critical edit operations are performed.

Even though we did not evaluate the semantic lifting approach in a long-term application, engineers at Siemens Mobility support the hypothesis that it can be useful for software product line engineering related tasks and quality assurance of models. Overall, we can conclude that the edit operations discovered by Ockham can be useful in the context of the case study in Sect. 2.

In Summary, Ockham is able to identify relevant edit operations that correspond to typical edit scenarios in the history of projects. The mining doesn’t scale to large connected model differences, so these have to be excluded from the mining. As a consequence of this, but also because a minimum frequency threshold has to be set, we can not claim that our approach is *complete* in the sense that it will discover all relevant edit operations. Furthermore, hard tasks for the subgraph miner typically lead also to bad results with respect to the correctness of the mined edit operations. The edit operations discovered by Ockham can also be applied in practical application scenarios, for example, to ease the analysis of large model differences by employing semantic lifting and filtering.

6.2 Further applications

We made an interesting observation when analyzing the distribution of the edit operations in the fourth experiment (see Fig. 4). Between some of the sample model differences, we observe high similarities of their distributions, while for others, there is almost no correlation. In other words, we can see clusters of the edit operation distributions, which can be seen more easily in Fig. 5.

A closer look at these clusters reveals that there is indeed a similarity in the model differences. For example, the cluster $[11-15, 17-23, 25-26]$ contains a lot of functional modifications of the models. For example, interfaces have been changed, so called Functional Addresses have been added, and activity diagrams were modified. The cluster $[24, 31-33]$ also contains functional modifications, but additionally documentation has changed a lot. For example, so called Reference Documents have been added, system requirements have been added or changed, and comments were added or changed. The cluster $[6-10, 27]$ contains a lot of added functionality but less modification to the documentation. Furthermore, the change profiles between different submodels for the same trainset type are often

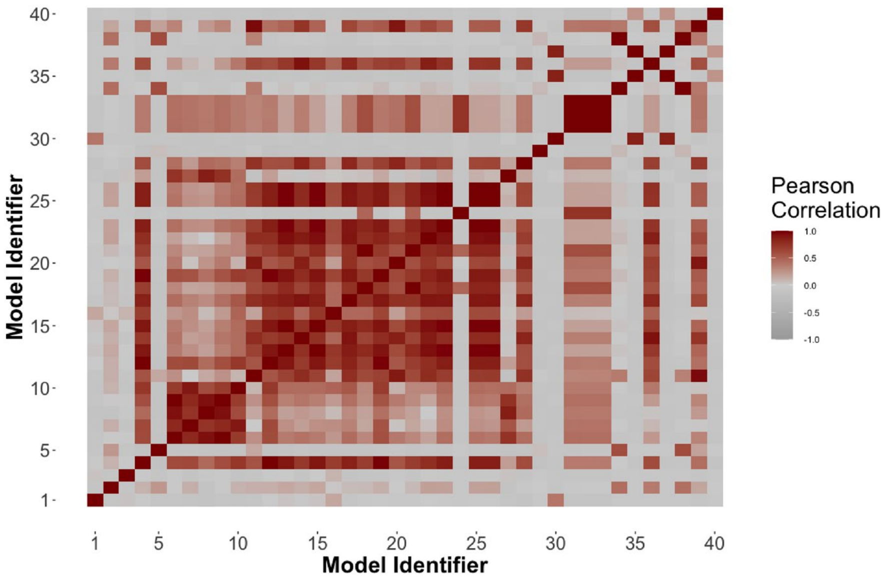


Fig. 5 Pearson correlation of the edit operation frequency distributions for 40 sample model differences. We have removed a dominant edit operation (Adding a Package) from the distributions to make the clusters appear more clearly in this visual representation. Qualitatively, the same clusters will be obtained though, without removing this edit operation from the frequency distribution

very similar. We can also find some model differences that are quite different from all other model differences and play some special role, for example, for cross-cutting functionality like test automation.

This suggests that the edit operations frequency distributions can be seen as a kind of *change profile* for the model difference, which can be used, for example, to gain further insights into the model differences. These change profiles can be used to reason about model differences or model evolution from a high-level perspective. Insights about the clusters can also be used, for example, to detect uncommon and possibly undesired changes. Since they are probability distributions, methods from statistics can be applied. Furthermore, they can also be seen as normed vectors (w.r.t. the 1-norm) and therefore also methods from linear algebra can be applied to further analyze model differences. Besides the analysis of the model evolution, the change profile can also be used for simulating model evolution, especially if different evolution scenarios are to be investigated. Similar profiles have already been used for model evolution simulation, for example, by Heider et al (2010), who refer to the profiles as evolution profiles. Another statistical approach to model simulation based on semantic lifting has been proposed by Shariat Yazdi et al (2016) but their edit operations have to be defined manually.

Dual to this cluster analysis, the change profiles can also be used to analyze co-occurrences of edit operations. For example, we observe a large Pearson correlation of ~ 0.99 between the edit operations ChangeUseCase and

ChangeActivity. This could also indicate that these edit operations can be combined to form a new, larger edit operation, which is the case in this concrete example. Some of the edit operations seem to be almost uncorrelated, for example, we observe that adding new functionality rarely occurs together with modifying existing functionality. Similar co-evolution analysis have been investigated in detail by Getir et al (2018). Note that Ockham is also a kind of co-change analysis that also takes structural similarities into account. The interplay between statistical co-occurrence analysis and structural graph mining in the analysis of model evolution is a promising future research direction.

6.3 Limitations

Locality relaxation: One limitation of our approach is the locality relaxation, which limits our ability to find patterns that are scattered across more than one connected component of the simple change graph. As we have seen in our railway case study, this usually leads to incomplete edit operations. Another typical example for violating the relaxation are naming conventions. In the future, we plan to use natural language processing techniques such as semantic matching to augment the models by further references.

No attribute information: For our experiments, we did not take attribute information into account. Attributes (e.g., the name of a component) could also be integrated into the edit operation as preconditions or to extract the parameters of an edit operation. For the purpose of summarizing a model difference or identifying violations in a model difference, preconditions and parameters are not important, though, but only the presence of structural patterns.

Application to simplified graphs: Generally, an edit operation is a model transformation. Model transformation engines such as Henshin provide features to deal with class inheritance or multi-object structures (roughly speaking, foreach loops in model transformations). In our approach, we are not leveraging these features yet. They could be integrated into Ockham in a post-processing step. For example, one possibility would be to feed the example instances of patterns discovered by Ockham into a traditional MTBE approach (Kehrer et al 2017).

Transient effects: We do not take so-called transient effects into account yet. One applied edit operation can invalidate the pre- or post-conditions of another edit operation. However, we have seen in our experiments that this only causes problems in cases where we apply only a few "correct" edit operations with high perturbation. In a practical scenario, the "perturbations" will more likely cancel each other out. When a transient effect occurs very frequently, a new pattern will be discovered. That is, when two (or more) operations are always applied together, we want to find the composite pattern, not the constituent ones.

Focus on single subgraphs instead of sets: Another limitation is the fact that we focused the optimization on *single* edit operations but not a *complete set* of edit operations. One could detect only the most-compressing edit operation and then substitute this in the model differences and re-run the mining to discover the second most-compressing edit operation and so on. Another solution would be to detect a set of candidate edit operations using Ockham and then select an optimal set using a meta-heuristic search algorithm optimizing the *total compression*. We leave this for further research.

Completeness of the set of mined edit operations: As we can see from experiment 4, a set of 30 edit operations is still far from being complete. In fact, with this set, we missed 70% of changes in our sample model differences. Anyway, the approach can also be applied iteratively, in the sense that edit operations which are already approved by domain experts can be used to substitute the corresponding subgraph by a single node. The graph mining can then be applied on these *contracted* graph databases until all changes are included in a change set. However, since a change does not unambiguously belong to a change set, this approach would be subject to the limitation from the previous paragraph.

6.4 Threats to validity

Internal validity: We have designed the first two experiments such that we can control input parameters of interest and observe their effect on the outcome. Ockham makes assumptions such as the locality relaxation, which could impair real-world applicability. Because of this and since we can not claim that the results from the first two experiments also hold true in a real-world setting, we additionally applied Ockham to an industrial case study. Our results increase our confidence that Ockham also gives reasonable results in a practical scenario.

In our simulations, we applied the edit operation randomly to a meta-model. To reduce the risk of observations that are only a result of this sampling, we created many example models. In the real-world setting, we compared the mined edit operations to random ones to rule out "patternicity" (Shermer 2008) as an explanation for high Likert rankings. None of our participants reported problems in understanding Henshin's visual notation, which gives us confidence regarding their judgements (despite for misconceptions). The participants of the interviews in the third experiment were also involved in the project where the model history was taken from. There might be the risk that the interviewees have only discovered operations they have "invented". In any case, because of the huge project size and because 22 out of 25 of the edit operations were recognized as typical by more than one of the participants, this is unlikely.

External validity: Some of the observations in our experiments could be due to the concrete set of edit operations in the example or even due to something in the meta-models. In the future, Ockham has to be tested for further meta-models to increase the external validity of our results. We have validated our approach in a real-world setting, which increases our confidence in its practicality, though. As can be seen from experiment 4, with the set of 30 edit operations, 70% of the changes are not part of a change set. It is therefore not yet clear, if the results regarding the compression of model differences also hold true when using larger sets of *meaningful* edit operations. Since we have used an exact subgraph miner, we can be sure that the discovered edit operation are independent of the subgraph mining algorithm.

7 Related work

Various approaches have been proposed to (semi-)automatically learn model transformations in the field of model transformation by example (MTBE). In the first systematic approach of MTBE, Varró (2006) proposes an iterative procedure that attempts to derive *exogenous* (i.e., source and target meta-model are different) model transformations by examples. Appropriate examples need to be provided for the algorithm to work. Many approaches to learning exogenous model transformations have been proposed until now. For example, Berramla et al (2020) use statistical machine translation and language models to derive transformations. Baki and Sahraoui (2016) apply simulated annealing to learn operations. Regarding *exogenous* transformations there is also an approach by Saada et al (2014), which uses graph mining techniques to learn concepts, which are then used to identify new transformation patterns.

As mentioned in the introduction, most closely related approach to ours is MTBE for *endogenous* model transformations. Compared to exogenous MTBE, there are only a few studies available for endogenous MTBE. Brosch et al (2009) present a tool called Operation Recorder, which is a semi-automatic approach to derive model transformations by recording all transformation steps. A similar approach is presented by Sun et al (2011), who also infer complex model transformations from a demonstration. Alshanqiti et al (2012) learn transformation rules from a set of examples by generalizing over pre- and postcondition graphs. Their approach has been applied to the derivation of edit operations, including negative application conditions and multi-object patterns by Kehrer et al (2017). Instead of learning a single operation, Mokaddem et al (2018) use a genetic algorithm to learn a set of refactoring rule pairs of examples before and after applying refactorings. The creation of candidate transformations that conform to the meta-model relies on a “fragment type graph”, which allows them to grow candidate patterns that conform to the meta-model. Their algorithm optimizes a model modification and preservation score. Ghannem et al (2018) also use a genetic algorithm (i.e., NSGA-II) to learn model refactorings from a set of “bad designed” and “good designed” models. Their approach distinguishes between structural similarity and semantic similarity and tries to minimize structural and semantic similarity between the initial model and the bad designed models and to maximize the similarity between the initial and the well designed models.

All of these approaches for learning endogenous model transformations are (semi-)supervised. Either a concrete example is given (which only contains the transformation to be learned) or a set of positive and negative examples is given. In the case of Mokaddem et al.’s genetic approach, it is assumed that all transformations that can be applied are actually applied to the source models. For the meta-model used in our real-world case study, we do not have any labeled data. In general, we are not aware of any fully unsupervised approach to learn endogenous model transformations. To reduce the search space, we leverage the evolution of the models in the model repository, though. We do not directly work on the models as in the approaches discussed above, but we work on structural model differences.

Regarding one of our motivations for mining edit operations, namely to simplify differences, there are several approaches in the source code domain (Yu et al 2011; Martinez et al 2013). These approaches are more comparable to the approach of semantic lifting (Kehrer et al 2011), to aggregate or filter model differences according to given patterns but they are not learning the patterns themselves. There are also approaches to mine change patterns in source code. For example, Dagit and Sottile (2013) propose an approach based on the abstract syntax tree, and Nguyen et al (2019) mine patterns based on a so called fine-grained program dependence graph. Janke and Mäder (2020) derive *fine-grained edit scripts* from abstract syntax tree differences and transform them to a graph representation for graph mining. There is also some work that focuses on mining design patterns from source code (Oruc et al 2016; Balanyi and Ferenc 2003; Ferenc et al 2005; Dong et al 2009). The idea behind these approaches — learning (change) patterns from a version history — is comparable to ours. In contrast to these approaches, Ockham works on a kind of abstract syntax graph, which already includes domain knowledge given by the metamodel. Furthermore, we do not use a similarity metric to detect change groups or frequent changes but use an (exact) subgraph mining approach. In model-driven engineering, one often has some kind of identifiers for the model elements, which makes the differencing more reliable and removes the need for similarity-based differencing methods.

8 Conclusion and outlook

We have proposed an approach, Ockham, for automatically deriving edit operations specified as in-place model transformations from model repositories. Ockahm is based on the idea that a meaningful edit operation will be one that provides a good compression for the model differences. In particular, it uses frequent subgraph mining on labeled graph representation of model differences to discover frequent patterns in the model differences. The patterns are then filtered and ranked based on a compression metric to obtain a list of recommendations for meaningful edit operations. To the best of our knowledge, Ockham is the first approach for learning domain-specific edit operations in a fully *unsupervised* manner, that is, without relying on any manual intervention or input from a developer or domain expert.

We have successfully evaluated Ockham in two controlled experiments using synthetic ground-truth EMF models and on a large-scale real-world case study in the railway domain. We found that Ockham is able to extract edit operations that have actually been applied before and that it discovers meaningful edit operations in a real-world setting. Including too large components in the difference graphs can adversely affect Ockham in discovering the applied edit operations. Performance mostly depends on the number of applied edit operations in a model difference. Furthermore, we have shown how the edit operations discovered by Ockham can be used in practical applications. For example, they can be used in a semantic lifting, to express model differences in terms of edit operations and therefore “compress” the model differences. Depending on the concrete task related to the model differences, the lifted model differences can then also be filtered for the edit operations of interest. The frequency distributions of edit operations in model differences can also be used for a high-level analysis of the model differences, for example, to discover hot spots, where a lot of functional evolution has happened. Ockham can be applied to models of any Domain-Specific Modeling Language for which model histories are available. New effective edit operations that are performed by the users can be learned at runtime and recommendations can be made.

For our future research, we plan to extend Ockham by a meta-heuristic search to identify the optimal *set* of operations. Extending state-of-the-art frequent subgraph miners to handle concepts like inheritance, multi-object structures, and other concepts from model-driven software development will very likely improve the results of Ockham. Since exact frequent subgraph mining is not tractable when large connected components of the difference graph have to be taken into account, heuristics and approximate algorithms have to be investigated. For example, one could investigate the effect of preprocessing the graph database using a clustering algorithm. Additionally, the effect of using approximate frequent subgraph miners directly on the database of simple change graphs might improve the scalability of the solution. Furthermore, to reduce the amount of recommended edit operations the results could be postprocessed using heuristics (e.g., that the edit operations occur steadily over the modelling history). An alternative approach, which we want to study in the future, is to use a clustering algorithm and then feed the clusters into the frequent subgraph mining step of our approach. This would allow us also to deal with examples in which the connected components are huge.

Acknowledgements The work of the second author has been partially supported by the German Research Foundation within the project VariantSync (KE 2267/1-1).

Funding Open Access funding enabled and organized by Projekt DEAL.

References

Acreţoaie, V., Störrle, H., Strüber, D.: VMTL: a language for end-user model transformation. Softw. Syst. Model. **17**(4), 1139–1167 (2018). https://doi.org/10.1007/s10270-016-0546-9

Alshanqiti, A.M., Heckel, R., Khan, T.A.: Learning minimal and maximal rules from observations of graph transformations. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **47**, 58–98 (2012). https://doi.org/10.14279/tuj.eceasst.58.848

Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the Eclipse Modeling Framework. In: Proceedings of the International Conference on Automated Software Engineering (ASE). IEEE/ACM, pp. 141–184, (2013) https://doi.org/10.1007/s10515-012-0114-7

Arendt, T., Biermann, E., Jurack, S., et al.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), Springer, pp. 121–135 (2010)

Avazpour, I., Grundy, J., Grunske, L.: Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations. J. Vis. Lang. Comput. **28**, 195–211 (2015). https://doi.org/10.1016/j.jvlc.2015.02.005

Baki, I., Sahraoui, H.: Multi-step learning and adaptive search for learning complex model transformations from examples. ACM Trans. Softw. Eng. Method. **25**(3), 1–36 (2016). https://doi.org/10.1145/2904904

Balanyi, Z., Ferenc, R.: Mining design patterns from C++ source code. In: Proceedings of the International Conference on Software Maintenance (ICSM). IEEE, pp. 305–314, (2003) https://doi.org/10.1109/ICSM.2003.1235436

Bariatti, F., Cellier, P., Ferré, S.: Graphmdl: Graph pattern selection based on minimum description length. In: Berthold MR, Feelders A, Krempl G (eds) Advances in Intelligent Data Analysis XVIII. Springer International Publishing, pp. 54–66, (2020) https://doi.org/10.1007/978-3-030-44584-3_5

ben Fadhel, A., Kessentini, M., Langer, P., et al.: Search-based detection of high-level model changes. In: Proceedings of the International Conference on Software Maintenance (ICSM). IEEE, pp. 212–221, (2012) https://doi.org/10.1109/ICSM.2012.6405274

Berramla, K., Deba, E.A., Wu, J., et al.: Model transformation by example with statistical machine translation. In: Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD), INSTICC. SciTePress, pp. 76–83 (2020)

Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Softw. Syst. Model. **11**(2), 227–250 (2012). https://doi.org/10.1007/s10270-011-0199-7

Brosch, P., Langer, P., Seidl, M., et al.: An example is worth a thousend words: Composite operation modeling by-example. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS). ACM, pp. 271–285, (2009) https://doi.org/10.1007/978-3-642-04425-0_20

Burdusel, A., Zschaler, S., Strüber, D.: MDEOptimiser: A search based model engineering tool. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings. ACM, pp. 12–16, (2018) https://doi.org/10.1145/3270112.3270130

Cook, D.J., Holder, L.B.: Mining Graph Data. Wiley (2006)

Dagit, J., Sottile, M.J.: Identifying change patterns in software history. In: Proceedings of the International workshop on Document Changes: Modeling, Detection, Storage and Visualization. CEUR-WS.org, (2013) arXiv:1307.1719

Djoko, S.: Substructure discovery using minimum description length principle and background knowledge. Proc. Natl. Conf. Artif. Intell. **2**, 1442 (1994)

Dong, J., Zhao, Y., Peng, T.: A review of design pattern mining techniques. Int. J. Softw. Eng. Knowl. Eng. **19**(6), 823–855 (2009)

Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Lecture Notes in Computer Science, pp. 161–177, (2004)

Ehrig, K., Ermel, C., Hänsgen, S., et al.: Generation of visual editors as Eclipse plug-ins. In: Proceedings of the International Conference on Automated Software Engineering (ASE). IEEE, pp. 134–143, (2005) https://doi.org/10.1145/1101908.1101930

Ferenc, R., Beszedes, A., Fulop, L., et al.: Design pattern mining enhanced by machine learning. In: Proceedings of the International Conference on Software Maintenance (ICSM). IEEE, pp. 295–304, (2005) https://doi.org/10.1109/ICSM.2005.40

Getir, S., Grunske, L., van Hoorn, A., et al.: Supporting semi-automatic co-evolution of architecture and fault tree models. J. Syst. Softw. **142**, 115–135 (2018) https://doi.org/10.18420/se2019-13

Ghannem, A., Kessentini, M., Hamdi, M.S., et al.: Model refactoring by example: a multi-objective search based software engineering approach. J. Softw. Evolut. Process **30**(4), 1–20 (2018) https://doi.org/10.1002/smr.1916

Grünwald, P.D., Grunwald, A.: The Minimum Description Length Principle. MIT Press (2007)

Hegedüs, Á., Horváth, Á., Ráth, I., et al.: Quick fix generation for DSMLs. In: Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, pp. 17–24, (2011) https://doi.org/10.1109/VLHCC.2011.6070373

Heider, W., Froschauer, R., Grünbacher, P., et al.: Simulating evolution in model-based product line engineering. Inf. Softw. Technol. **52**(7), 758–769 (2010). https://doi.org/10.1016/j.infsof.2010.03.007

Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: Software Language Engineering. ACM, pp. 163–182, (2010) https://doi.org/10.5555/1964571.1964585

Hölldobler, K., Rumpe, B., Weisemöller, I.: Systematically deriving domain-specific transformation languages. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS). ACM/IEEE, pp. 136–145, (2015) https://doi.org/10.5555/3351736.3351756

Janke, M., Mäder, P.: Graph based mining of code change patterns from version control commits. Trans. Softw. Eng. (2020). https://doi.org/10.1109/TSE.2020.3004892

Jiang, C., Coenen, F., Zito, M.: A survey of frequent subgraph mining algorithms. Knowl. Eng. Rev. **28**(1), 75–105 (2013). https://doi.org/10.1017/S0269888912000331

Kahani, N., Bagherzadeh, M., Cordy, J.R., et al.: Survey and classification of model transformation tools. Softw. Syst. Model. **18**(4), 2361–2397 (2019). https://doi.org/10.1007/s10270-018-0665-6

Kappel, G., Langer, P., Retschitzegger, W., et al.: Model transformation by-example: A survey of the first wave. In: Conceptual Modelling and Its Theoretical Foundations - Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday. Springer, pp. 197–215, (2012) https://doi.org/10.1007/978-3-642-28279-9_15

Kehrer, T.: Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering. PhD thesis, University of Siegen (2015)

Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Proceedings of the International Conference on Automated Software Engineering (ASE). ACM/IEEE, pp. 163–172, (2011) https://doi.org/10.1109/ASE.2011.6100050

Kehrer, T., Kelter, U., Ohrndorf, M., et al.: Understanding model evolution through semantically lifting model differences with SiLift. In: Proceedings of the International Conference on Software Maintenance (ICSM). IEEE, pp. 638–641, (2012a) https://doi.org/10.1109/ICSM.2012.6405342

Kehrer, T., Kelter, U., Pietsch, P., et al.: Adaptability of model comparison tools. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, IEEE, pp. 306–309, (2012b) https://doi.org/10.1145/2351676.2351731

Kehrer, T., Rindt, M., Pietsch, P., et al.: Generating edit operations for profiled uml models. In: MoDELS Workshop on Models and Evolution (ME@MoDELS), Citeseer, pp. 30–39, (2013)

Kehrer, T., Taentzer, G., Rindt, M., et al.: Automatically deriving the specification of model editing operations from meta-models. In: Proceedings of the International Conference on Model Transformations (ICMT), pp. 173–188, (2016) https://doi.org/10.1007/978-3-319-42064-6_12

Kehrer, T., Alshanqiti, A.M., Heckel, R.: Automatic inference of rule-based specifications of complex in-place model transformations. In: Proceedings of the International Conference on Model Transformations (ICMT). Springer, pp. 92–107, (2017) https://doi.org/10.1007/978-3-319-61473-1_7

Ketkar, N.S., Holder, L.B., Cook, D.J.: Subdue: Compression-based frequent pattern discovery in graph data. In: Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations. ACM, p 71-76, (2005) https://doi.org/10.1145/1133905.1133915

Khelladi, D.E., Hebig, R., Bendraou, R., et al.: Detecting complex changes and refactorings during (meta) model evolution. Inf. Syst. **62**, 220–241 (2016). https://doi.org/10.1016/j.is.2016.05.002

Kolovos, D.S., Di Ruscio, D., Pierantonio, A., et al.: Different Models for Model Matching: An analysis of approaches to support model differencing. In: Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models, IEEE, pp. 1–6, (2009) https://doi.org/10.1109/CVSM.2009.5071714

Kolovos, D.S., Rose, L.M., Abid, S.B., et al.: Taming EMF and GMF using model transformation. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), Springer, pp. 211–225,(2010) https://doi.org/10.5555/1926458.1926479

Kögel, M., Helming, J., Seyboth, S.: Operation-based conflict detection and resolution. In: Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models, IEEE, pp. 43–48, (2009) https://doi.org/10.1109/CVSM.2009.5071721

Kögel, S., Groner, R., Tichy, M.: Automatic change recommendation of models and meta models based on change histories. In: Proceedings of the 10th Workshop on Models and Evolution co-located with

Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS). ACM/IEEE, pp. 14–19 (2016)
Langer, P., Wimmer, M., Brosch, P., et al.: A posteriori operation detection in evolving software models. J. Syst. Softw. **86**(2), 551–566 (2013). https://doi.org/10.1016/j.jss.2012.09.037
Martinez, M., Duchien, L., Monperrus, M.: Automatically extracting instances of code change patterns with AST analysis. In: Proceedings of the International Conference on Software Maintenance (ICSM). IEEE, pp. 388–391, (2013) https://doi.org/10.1109/ICSM.2013.54
Mazanek, S., Minas, M.: Generating correctness-preserving editing operations for diagram editors. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **18**, 58 (2009). https://doi.org/10.14279/tuj.eceasst.18.262
Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**(1–2), 125–142 (2006). https://doi.org/10.1016/j.entcs.2005.10.021
Mokaddem, C., Sahraoui, H., Syriani, E.: Recommending model refactoring rules from refactoring examples. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS). ACM, pp.257–266, (2018)https://doi.org/10.1145/3239372.3239406
Nguyen, H.A., Nguyen, T.N., Dig, D., et al.: Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM/IEEE, pp. 819–830 (2019)
Nijssen, S., Kok, J.N.: The Gaston tool for frequent subgraph mining. Electron. Notes Theor. Comput. Sci. **127**(1), 77–87 (2005). https://doi.org/10.1016/j.entcs.2004.12.039
Ohrndorf, M., Pietsch, C., Kelter, U., et al.: ReVision: A tool for history-based model repair recommendations. In: Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings. ACM, pp. 105–108, (2018) https://doi.org/10.1145/3419017
Oruc, M., Akal, F., Sever, H.: Detecting design patterns in object-oriented design models by using a graph mining approach. In: Proceedings of the International Conference in Software Engineering Research and Innovation (CONISOFT). IEEE, pp. 115–121,(2016) https://doi.org/10.1109/CONISOFT.2016.26
Pietsch, P., Yazdi, H.S., Kelter, U.: Generating realistic test models for model processing tools. In: Proceedings of the International Conference on Automated Software Engineering (ASE). IEEE, pp. 620–623, (2011) https://doi.org/10.1109/ASE.2011.6100140
Polanyi, M.: Personal Knowledge: Towards a Post Critical Philosophy. University of Chicago Press, Chicago (1958)
Rodrigues Da Silva, A.: Model-driven engineering: a survey supported by the unified conceptual model. Comput. Lang. Syst. Struct. **43**, 139–155 (2015). https://doi.org/10.1016/j.cl.2015.06.001
Rose, L.M., Herrmannsdoerfer, M., Mazanek, S., et al.: Graph and model transformation tools for model migration: empirical results from the transformation tool contest. Softw. Syst. Model. **13**(1), 323–359 (2014)
Saada, H., Huchard, M., Liquiere, M., et al.: Learning model transformation patterns using graph generalization. In: Proceedings of the International Conference on Concept Lattices and Their Applications. CEUR-WS.org, pp. 11–22 (2014)
Schmidt, M., Gloetzner, T.: Constructing difference tools for models using the SiDiff framework. In: Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings. ACM/IEEE, pp. 947–948, (2008) https://doi.org/10.1145/1370175.1370201
Schmidt, M., Wenzel, S., Kehrer, T., et al.: History-based merging of models. In: ICSE Workshop on Comparison and Versioning of Software Models (CVSM), IEEE, pp. 13–18 (2009)
Schröder, G., Thiele, M., Lehner, W.: Setting goals and choosing metrics for recommender system evaluations. In: Proceedings of the Conference on Recommender Systems (RecSys). ACM, p 53 (2011)
Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003). https://doi.org/10.1109/MS.2003.1231150
Shariat Yazdi, H., Angelis, L., Kehrer, T., et al.: A framework for capturing, statistically modeling and analyzing the evolution of software models. J. Syst. Softw. **118**, 176–207 (2016). https://doi.org/10.1016/j.jss.2016.05.010
Shermer, M.: Patternicity: finding meaningful patterns in meaningless noise. Sci. Am. **299**(5), 48 (2008). https://doi.org/10.1038/scientificamerican1208-48
Siegmund, J., Siegmund, N., Apel, S.: Views on internal and external validity in empirical software engineering. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, pp. 9–19, (2015) https://doi.org/10.5555/2818754.2818759

Stephan, M., Cordy, J.R.: A survey of model comparison approaches and applications. In: Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 265–277,(2013) https://doi.org/10.5220/0004311102650277

Sun, Y., Gray, J., White, J.: MT-Scribe: An end-user approach to automate software model evolution. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM/IEEE, pp. 980–982, (2011) https://doi.org/10.1145/1985793.1985966

Taentzer, G., Crema, A., Schmutzler, R., et al.: Generating domain-specific model editors with complex editing commands. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE), Springer, pp. 98–103, (2007)

Tinnes, C., Kehrer, T., Mitchell, J., et al.: Learning domain-specific edit operations from model repositories with frequent subgraph mining. In: Proceedings of the International Conference on Automated Software Engineering (ASE). ACM/IEEE, (2021) https://doi.org/10.1109/ASE51524.2021.9678698

Van Deursen, A., Visser, E., Warmer, J.: Model-driven software evolution: A research agenda. Technical Report Series TUD-SERG-2007-006 (2007) https://doi.org/10.1109/MSR.2013.6624031

Varró, D.: Model transformation by example. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS). Springer, pp. 410–424, (2006)

Welke, P., Seiffarth, F., Kamp, M., et al.: HOPS: Probabilistic subtree mining for small and large graphs. In: Proceedings of the Conference on Knowledge Discovery (KDD). ACM, pp. 1275–1284, (2020) https://doi.org/10.1145/3394486.3403180

Yan, X., Han, J.: Closegraph: mining closed frequent graph patterns. In: Proceedings of the Conference on Knowledge Discovery (KDD), pp. 286–295, (2003) https://doi.org/10.1145/956750.956784

Yu, Y., Tun, T.T., Nuseibeh, B.: Specifying and detecting meaningful changes in programs. In: Proceedings of the International Conference on Automated Software Engineering (ASE). IEEE, pp. 273–282, (2011) https://doi.org/10.1109/ASE.2011.6100063

Authors and Affiliations

Christof Tinnes[1,3] · Timo Kehrer[2,4] · Mitchell Joblin[1,3] · Uwe Hohenstein[1] · Andreas Biesdorf[1,5] · Sven Apel[1,6]

Timo Kehrer
timo.kehrer@inf.unibe.ch

Mitchell Joblin
mitchell.joblin@siemens.com

Uwe Hohenstein
uwe.hohenstein@siemens.com

Andreas Biesdorf
andreas.biesdorf@siemens.com

Sven Apel
apel@cs.uni-saarland.de

[1] Siemens AG, München, Germany

[2] Humboldt-Universität zu Berlin, Berlin, Germany

[3] Saarland University, Saarbrücken, Germany

[4] University of Bern, Bern, Switzerland

[5] Trier University of Applied Sciences, Trier, Germany

[6] Saarland Informatics Campus, Saarbrücken, Germany