Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining

Christof Tinnes†, Timo Kehrer*, Mitchell Joblin‡†, Uwe Hohenstein†, Andreas Biesdorf†, Sven Apel‡
†Siemens AG – Technology, München, Germany
*Humboldt-Universität zu Berlin, Berlin-Adlershof, Germany
‡Saarland University, Informatics Campus, Saarbrücken, Germany

***Abstract*—Model transformations play a fundamental role in model-driven software development. They can be used to solve or support central tasks, such as creating models, handling model co-evolution, and model merging. In the past, various (semi-)automatic approaches have been proposed to derive model transformations from meta-models or from examples. 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 to practitioners in an industrial setting.**

I. 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) [54], 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 [60].

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 [66] 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 [24, 41, 50], model editors [19, 65], operation-based merging [40, 58], model refactoring [4, 17], model optimization [12], meta-model evolution and model co-evolution [3, 21, 25, 43, 55], semantic lifting of model differences [8, 33, 35, 39, 44], model generation [52], 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 [17, 30, 32]. Secondly, for some tasks, domain-specific transformations are a form of tacit knowledge [53], 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 [5] and transformation languages derived from a modeling language's concrete syntax [1, 26] 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 [36, 37, 46]. 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) [11, 30, 64], 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 [17, 32] 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. [32], 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. [17] 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. Section II). 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.

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 is scales to real-world repositories.

II. 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 [27] 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/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. 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 [4, 8, 21, 33, 39, 41, 44, 50, 65].

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. [35], which uses a set of edit operations as configuration input, can be used but the approach requires the edit operations to be defined already. These large model differences 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 Section V.

III. BACKGROUND

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

A. 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 1 illustrates how a simplified excerpt from an architectural model of our case study from Section II 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 [10], 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

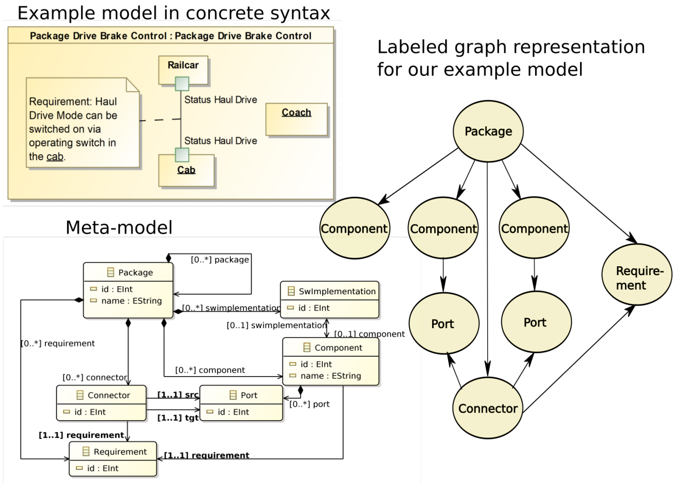

Fig. 1. We consider models as labeled graphs, where labels represent types of nodes and edges defined by a meta model. For the sake of brevity, types of edges are omitted in the figure.

of nodes, E is a subset of $V \times V$, called the edge set, and $\lambda : V \cup E \rightarrow L$ 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 an induced 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. $\{(v, v_1), (v_1, v_2), \ldots, (v_n, u)\} \subseteq E_C \; \cup \tilde{E}_C$, where $\tilde{E}_C$ is the set of all reversed edges, that is, $(u, v) \in E_C$ becomes $(v, u) \in \tilde{E}_C$.

B. 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 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 [28]. A general introduction to graph mining is given by Cook and Holder [13], who also proposed a compression-based subgraph miner called SUBDUE [38]. SUBDUE has also been one of our main inspirations for a compression-based approach. OCKHAM is based on GASTON [49], which mines frequent subgraphs by first focusing on frequent paths, then extending to frequent trees, and finally extending the trees to cyclic graphs.

C. 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 Visser et al. [66], edit operations can describe regular evolution [66], 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 [47], 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 [3] supports the specification of in-place model transformations in a declarative manner. It is based on graph transformation concepts [18], and it provides a visual language for the definition of transformation rules, which is used, for example, in the last step of Figure 2. Roughly speaking, transformation rules specify graph patterns that are to be found and created or deleted.

IV. APPROACH

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

OCKHAM consists of the five steps illustrated with a running example in Figure 2. 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. Section III).

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 [31]. First, the corresponding model elements in the model graphs G_n and G_{n+1} are determined using a model matcher [42]. 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.

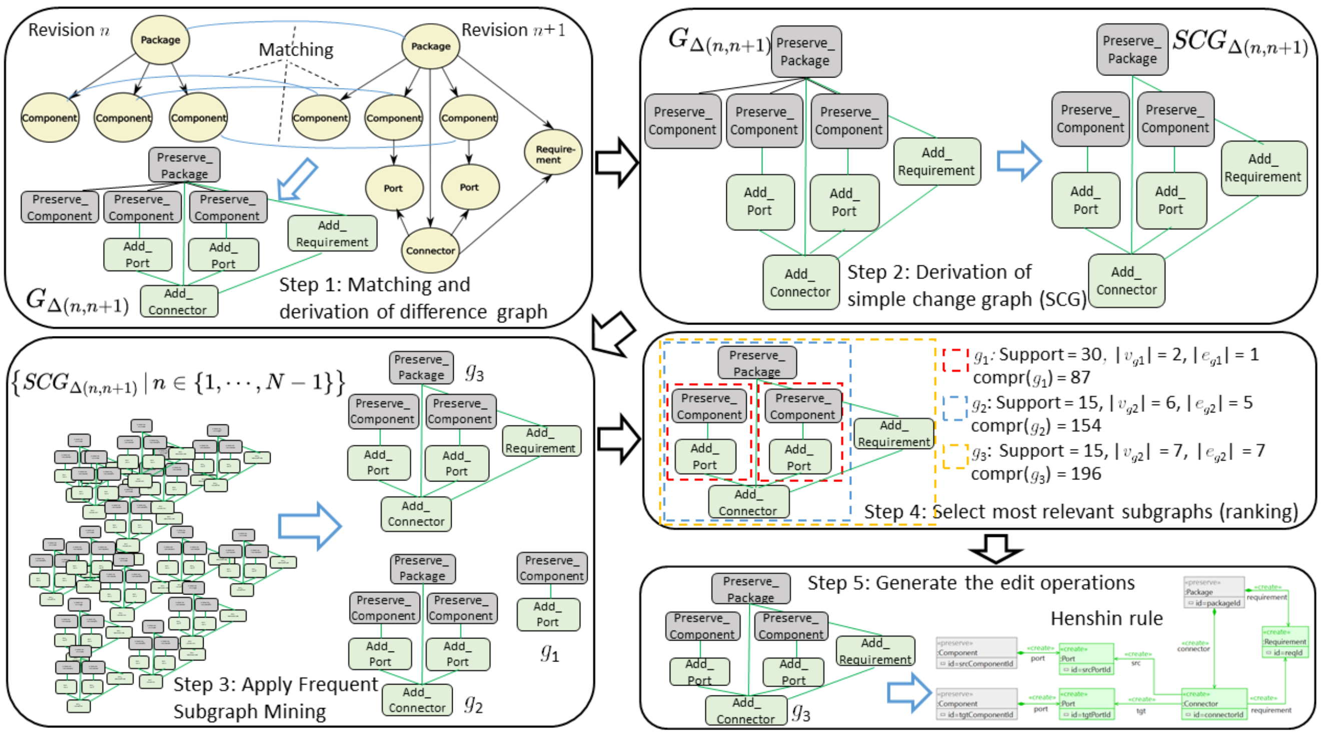

Fig. 2. The 5-step process for mining edit operations with OCKHAM.

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* [50]. 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 version n to $n+1$ appear only once in $G_{\Delta(n,n+1)}$ (indicated by the label prefix "preserved_"), 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 "delete_" and "create_").

For illustration, assume that the architectural model shown in Figure 1 is the revised version $n+1$ of a version n by adding the ports along with the connector and its associated requirement. Figure 2 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 Figure 2. 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 [34, 57] 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 Section II. 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. [42]; a survey of model comparison approaches is given by Stephan and Cordy [63].

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 Section VI-B. 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 "delete_*" and "create_*", 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, these preserved nodes 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 [31]. The derivation of a simple change graph from a given difference graph is illustrated in the second step of Figure 2. 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 Figure 2.

Step 3: Apply Frequent Connected Subgraph Mining: When we apply the first two steps to a model history, we obtain a set of simple change graphs $\{SCG_{\Delta(n,n+1)} \mid n \in \{1, \ldots, N-1\}\}$, 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. 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 Section VI. We run the frequent subgraph miner for the threshold found via the approximate tree miner. Step 3 of Figure 2 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 [49] 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 48h). GASTON (with embedding lists) was able to terminate in less then 10s 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 [38], which directly tries to optimize compression. Furthermore, SUBDUE was not able to discover both edit operations in the second experiment (see Section V), 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 [68] 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 [23], the most relevant patterns should not be the most frequent ones but the ones that give us a maximum compression for our original data [15]. 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 Section V to validate whether this assumption holds. We define the compression value by $\mathrm{compr}(g) = (\mathrm{supp}(g) - 1) \cdot (|V_g| + |E_g|)$, 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. 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 [59].

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

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

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

For our running example in Step 4 of Figure 2, 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* Python library.

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 our learned edit operations. As illustrated in Step 5 of Figure 2, the subgraphs can be transformed to Henshin transformation rules in a straightforward manner. We use HENSHIN because it is used for the semantic lifting approach in our case study from Sec. II. 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. [29].

V. EVALUATION

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

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

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.

B. Experiment Setup

We conduct three 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 Section II 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 [62], but the combination of all experiments is suitable to assess whether OCKHAM can discover relevant edit operations.

We run the experiments on an Intel® Core™ 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 Figure 1.

For this experiment, we only apply one kind of edit operation (the one from our running example in Figure 2) 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 $m_0 \to m_1 \to \dots m_{d-1} \to m_d$. Each evolution step $m_i \to m_{i+1}$ 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 [10] to apply model transformations to one model revision. We then build the difference of two successive models as outlined in Section IV. 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 [59]:

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

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 extend 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 Section II). 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.

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. 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.

C. Results

Experiment 1: In Table I, we list the MAP@k scores for all datasets in the experiment. Table III 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 1.17s per dataset.

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

TABLE I
THE MAP@K SCORES FOR THE RESULTS USING COMPRESSION AND FREQUENCY FOR 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 II
THE MAP@K SCORES FOR THE RESULTS USING COMPRESSION AND FREQUENCY 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 III
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

TABLE IV
THE SPEARMAN CORRELATION MATRIX FOR EXPERIMENT 2.

	p	Size at threshold	Mining time	e	Mean #Nodes per comp
AP	-0.31	-0.05	-0.25	-0.07	-0.19
p	–	0.20	0.27	0	0.30
Size at Threshold	–	–	0.53	0.51	0.58
Mining Time	–	–	–	0.87	0.92
e	–	–	–	–	0.92

Experiment 3: Table V 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 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 V.

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 higher than the mean for the random operations. We can therefore reject the null hypothesis.

TABLE V
STATISTICS FOR THE LIKERT VALUES OF MINED AND RANDOM EDIT OPERATIONS FOR EXPERIMENT 3.

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

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 extended" (see also Section VI). Some of the edit operations found by OCKHAM, but not recognized by the participants, where 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 Figure 2. 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.

VI. DISCUSSION

A. 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.

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 a mean Likert score of almost 4.1 shows. 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. 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 III, we observe that increasing the number of edit operations has a negative effect on the average precision.

TABLE VI
THE MAIN DRIVERS FOR OCKHAM TO FAIL IN DETECTING THE CORRECT SUBGRAPH IN EXPERIMENT 1.

	p	Mean #Nodes per comp	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

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 IV, 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 weaker.

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 VI. 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 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.

TABLE VII
POSSIBLE DRIVERS FOR A LOW RANK (≥ 5).

d	e	p	Mean #Nodes per comp	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

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 VII, 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 III, we ca 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 III). This actually means that large mining times usually come with a bad ranking. The same effect can be observed for Experiment 2 (Table IV). We can also see, that the mining time correlates with the size at threshold.

B. 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 [32].

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.

C. 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" [61] 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. Since we have used an exact subgraph miner, we can be sure that the discovered edit operation are independent of the subgraph mining algorithm.

VII. 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 [67] 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. [9] use statistical machine translation and language models to derive transformations. Baki and Sahraoui [6] apply simulated annealing to learn operations. Regarding *exogenous* transformations there is also an approach by Saada et al. [56], 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. [11] 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 Yun et al. [64], who also infer complex model transformations from a demonstration. Alshanqiti et al. [2] 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 [32]. Instead of learning a single operation, Mokaddem et al. [17] 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. [22] 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 [45, 69]. These approaches are more comparable to the approach of semantic lifting [35], 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 et al. propose an approach based on the abstract syntax tree [14], and Nguyen et al. mine patterns based on a so called fine-grained program dependence graph [48]. There is also some work that focuses on mining design patterns from source code [7, 16, 20, 51]. 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 meta-model. 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.

VIII. 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. 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. 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 of the difference graph are huge.

Acknowledgments

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

References

[1] Vlad Acreţoaie, Harald Störrle, and Daniel Strüber. VMTL: A language for end-user model transformation. *Software & Systems Modeling*, 17(4):1139–1167, 2018.

[2] Abdullah M. Alshanqiti, Reiko Heckel, and Tamim Ahmed Khan. Learning minimal and maximal rules from observations of graph transformations. *Electronic Communication of the European Association of Software Science and Technology*, 47, 2012.

[3] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 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)*, pages 121–135. Springer, 2010.

[4] Thorsten Arendt and Gabriele Taentzer. A tool environment for quality assurance based on the Eclipse Modeling Framework. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 141–184. IEEE/ACM, 2013.

[5] Iman Avazpour, John Grundy, and Lars Grunske. Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations. *Journal of Visual Languages and Computing*, 28:195–211, 2015.

[6] Islem Baki and Houari Sahraoui. Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Transactions on Software Engineering and Methodology*, 25(3):1–36, 2016.

[7] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from C++ source code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 305–314. IEEE, 2003.

[8] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. Search-based detection of high-level model changes. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 212–221. IEEE, 2012.

[9] Karima Berramla., El Abbassia Deba., Jiechen Wu., Houari Sahraoui., and Abou Benyamina. Model transformation by example with statistical machine translation. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 76–83. INSTICC, SciTePress, 2020.

[10] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and Systems Modeling*, 11(2):227–250, 2012.

[11] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. 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)*, pages 271–285. ACM, 2009.

[12] Alexandru Burdusel, Steffen Zschaler, and Daniel Strüber. MDEOptimiser: A search based model engineering tool. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings*, pages 12–16. ACM, 2018.

[13] Diane J Cook and Lawrence B Holder. *Mining graph data*. John Wiley & Sons, 2006.

[14] Jason Dagit and Matthew J. Sottile. Identifying change patterns in software history. In *Proceedings of the International workshop on Document Changes: Modeling, Detection, Storage and Visualization*. CEUR-WS.org, 2013.

[15] Surnjani Djoko. Substructure discovery using minimum description length principle and background knowledge. *Proceedings of the National Conference on Artificial Intelligence*, 2:1442, 1994.

[16] Jing Dong, Yajing Zhao, and Tu Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(6):823–855, 2009.

[17] Chihab eddine Mokaddem, Houari Sahraoui, and Eugene Syriani. Recommending model refactoring rules from refactoring examples. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 257–266. ACM, 2018.

[18] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In *Lecture Notes in Computer Science*, volume 3256, pages 161–177, 2004.

[19] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as Eclipse plug-ins. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 134–143. IEEE, 2005.

[20] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele. Design pattern mining enhanced by machine learning. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 295–304. IEEE, 2005.

[21] Sinem Getir, Lars Grunske, André van Hoorn, Timo Kehrer, Yannic Noller, and Matthias Tichy. Supporting semi-automatic co-evolution of architecture and fault tree models. *Journal of Systems and Software*, 142:115–135, 2018.

[22] Adnane Ghannem, Marouane Kessentini, Mohammad Salah Hamdi, and Ghizlane El Boussaidi. Model refactoring by example: A multi-objective search based software engineering approach. *Journal of Software: Evolution and Process*, 30(4):1–20, 2018.

[23] Peter D Grünwald and Abhijit Grunwald. *The minimum description length principle*. MIT press, 2007.

[24] Ábel Hegedüs, Ákos Horváth, István Ráth, Moisés Castelo Branco, and Dániel Varró. Quick fix generation for DSMLs. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 17–24. IEEE, 2011.

[25] Markus Herrmannsdoerfer, Sander Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Software Language Engineering*, pages 163–182. ACM, 2010.

[26] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically deriving domain-specific transformation languages. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 136–145. ACM/IEEE, 2015.

[27] No Magic Inc. MagicDraw user guide, 2021.

[28] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *Knowledge Engineering Review*, 28(1):75–105, 2013.

[29] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, 18(4):2361–2397, 2019.

[30] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. 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*, pages 197–215. Springer, 2012.

[31] Timo Kehrer. *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.

[32] Timo Kehrer, Abdullah M Alshanqiti, and Reiko Heckel. Automatic inference of rule-based specifications of complex in-place model transformations. In *Proceedings of the International Conference on Model Transformations (ICMT)*, pages 92–107. Springer, 2017.

[33] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. Understanding model evolution through semantically lifting model differences with SiLift. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 638–641. IEEE, 2012.

[34] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. Adaptability of model comparison tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 306–309. IEEE, 2012.

[35] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 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)*, pages 163–172. ACM/IEEE, 2011.

[36] Timo Kehrer, Michaela Rindt, Pit Pietsch, and Udo Kelter. Generating edit operations for profiled uml models. In *MoDELS Workshop on Models and Evolution (ME@MoDELS)*, pages 30–39. Citeseer, 2013.

[37] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. Automatically deriving the specification of model editing operations from meta-models. In *Proceedings of the International Conference on Model Transformations (ICMT)*, volume 9765, pages 173–188, 2016.

[38] Nikhil S. Ketkar, Lawrence B. Holder, and Diane J. Cook. 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*, page 71–76. ACM, 2005.

[39] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. Detecting complex changes and refactorings during (meta)model evolution. *Information Systems*, 62:220–241, 2016.

[40] Maximilian Koegel, Jonas Helming, and Stephan Seyboth. Operation-based conflict detection and resolution. In *Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models*, pages 43–48. IEEE, 2009.

[41] Stefan Kögel, Raffaela Groner, and Matthias Tichy. 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)*, pages 14–19. ACM/IEEE, 2016.

[42] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard Paige. 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*, pages 1–6. IEEE, 2009.

[43] Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. Taming EMF and GMF using model transformation. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 211–225. Springer, 2010.

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

[45] Matias Martinez, Laurence Duchien, and Martin Monperrus. Automatically extracting instances of code change patterns with AST analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 388–391. IEEE, 2013.

[46] Steffen Mazanek and Mark Minas. Generating correctness-preserving editing operations for diagram editors. *Electronic Communication of the European Association of Software Science and Technology*, 18, 2009.

[47] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152(1-2):125–142, 2006.

[48] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 819–830. ACM/IEEE, 2019.

[49] Siegfried Nijssen and Joost N. Kok. The Gaston tool for frequent subgraph mining. *Electronic Notes in Theoretical Computer Science*, 127(1):77–87, 2005.

[50] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. ReVision: A tool for history-based model repair recommendations. In *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*, pages 105–108. ACM, 2018.

[51] Murat Oruc, Fuat Akal, and Hayri Sever. 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)*, pages 115–121. IEEE, 2016.

[52] Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. Generating realistic test models for model processing tools. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 620–623. IEEE, 2011.

[53] Michael Polanyi. *Personal Knowledge: Towards a Post Critical Philosophy*. University of Chicago Press, 1958.

[54] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems and Structures*, 43:139–155, 2015.

[55] Louis M. Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp, Sebastian Buchwald, Tassilo Horn, Elina Kalnina, Andreas Koch, Kevin Lano, Bernhard Schätz, and Manuel Wimmer. Graph and model transformation tools for model migration: Empirical results from the transformation tool contest. *Software & Systems Modeling*, 13(1):323–359, 2014.

[56] Hajer Saada, Marianne Huchard, Michel Liquiere, and Clémentine Nebut. Learning model transformation patterns using graph generalization. In *Proceedings of the International Conference on Concept Lattices and Their Applications*, pages 11–22. CEUR-WS.org, 2014.

[57] Maik Schmidt and Tilman Gloetzner. Constructing difference tools for models using the SiDiff framework. In *Proceedings of the International Conference on Software Engineering (ICSE): Companion Proceedings*, pages 947–948. ACM/IEEE, 2008.

[58] Maik Schmidt, Sven Wenzel, Timo Kehrer, and Udo Kelter. History-based merging of models. In *ICSE Workshop on Comparison and Versioning of Software Models (CVSM)*, pages 13–18. IEEE, 2009.

[59] Gunnar Schröder, Maik Thiele, and Wolfgang Lehner. Setting goals and choosing metrics for recommender system evaluations. In *Proceedings of the Conference on Recommender Systems (RecSys)*, page 53. ACM, 2011.

[60] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003.

[61] Michael Shermer. Patternicity: Finding meaningful patterns in meaningless noise. *Scientific American*, 299(5):48, 2008.

[62] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 9–19. IEEE, 2015.

[63] Matthew Stephan and James R Cordy. A survey of model comparison approaches and applications. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 265–277, 2013.

[64] Yu Sun, Jeff Gray, and Jules White. MT-Scribe: An end-user approach to automate software model evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 980–982. ACM/IEEE, 2011.

[65] Gabriele Taentzer, André Crema, René Schmutzler, and Claudia Ermel. Generating domain-specific model editors with complex editing commands. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, pages 98–103. Springer, 2007.

[66] Arie Van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. *Technical Report Series TUD-SERG-2007-006.*, 2007.

[67] Dániel Varró. Model transformation by example. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 410–424. Springer, 2006.

[68] Pascal Welke, Florian Seiffarth, Michael Kamp, and Stefan Wrobel. HOPS: Probabilistic subtree mining for small and large graphs. In *Proceedings of the Conference on Knowledge Discovery (KDD)*, pages 1275–1284. ACM, 2020.

[69] Yijun Yu, Thein Than Tun, and Bashar Nuseibeh. Specifying and detecting meaningful changes in programs. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 273–282. IEEE, 2011.