# Tackling Model Drifts in Industrial Model-driven Software Product Lines by Means of a Graph Database

Christof Tinnes[1], Uwe Hohenstein[1], Wolfgang Rössler[2] and Andreas Biesdorf[1]
[1]*T SSP ADM, Siemens AG, Otto-Hahn-Ring 6, 81730 Munich, Germany*
[2]*SMO RS EN CCIP AR, Siemens AG, Siemenspromenade 4, 91058 Erlangen, Germany*

Keywords: Model-driven Approach, MagicDraw, Graph Database, Neo4j.

Abstract: This paper reports on our experience of using a graph database to efficiently compare very large models in an industrial model-driven engineering project. The need for a comparison results from the fact that architectural models are reused. They conform to a common domain-specific language but diverge as they belong to different products managed in separate branches of a repository in the sense of a clone-and-own approach. In the presented industry project, huge models are developed and reside in the commercial tool MAGICDRAW. In fact, unlike many other tools, MAGICDRAW turned out to be capable to handle those huge models in industrial environments. In this context, there is a strong necessity to detect and judge relevant differences between models in different branches in order to avoid a model drift and loosing reuse opportunities across the products. Indeed, MAGICDRAW has a built-in difference tool, which however exposes an excessive number of differences, only a fraction of which are really relevant for certain tasks. We show that the capabilities of the graph database NEO4J can be leveraged to reduce the differences to relevant ones. The expressiveness of NEO4J turned out to be sufficient, just as the performance did.

## 1 INTRODUCTION

As software and systems are becoming increasingly complex, methodologies, paradigms, and general reuse concepts (Krueger, 1993) have been developed to handle the increasing complexity. One applied approach is Model-Driven Engineering (MDE) (Rodrigues Da Silva, 2015), which uses higher-level models to abstract from concrete software. Models describe and document software at a higher abstraction level and are also used to generate documentation in certain formats and (parts of the) source code. Another reuse concept is Software Product Line Engineering (SPLE). The idea of SPLE is to manage a family of products and reuse artifacts across different products of the family (Pohl et al., 2005). The artifacts can be source code, but other artifacts such as models can also be subject to reuse in the software product line (Batoryayer, 2004).

In this paper, we report on challenges in a real industrial project within our company. The project has set up a model-driven product line in the railway domain in which huge architectural models, described in a SysML-like domain-specific language (DSL), are the primary artifact. About 200 engineers use the tool MAGICDRAW (No-Magic-Inc., 2021) to specify architectural models in a graphical manner. Figure 1 shows one of several hundreds of modeling artefacts.

Since the product line has been organically grown, starting with one product and adding product by product over time, no platform approach with built-in variability support has been applied. Further reasons are that not all potential products are known in advance, the number of variants is expected to stay small, and the development teams are organized product-wise, not feature-wise. Consequently, the overhead for establishing and using a platform will not pay off (Pohl et al., 2005). Instead, a managed cloning approach (Dubinsky et al., 2013; Rubin et al., 2013) is followed: The common baseline of all products constitutes the main branch in a software versioning system, and products are kept in specific branches, initially copying the common baseline for reuse. This means that product branches are evolving independently of the baseline over time; changes to common parts are merged into the baseline or from the baseline to specific products from time to time. However, the independent evolution of models – often performed by different people – leads to a divergence of the common parts in the product branches. In particular, we noticed that certain important activities such as quality assurance, change



Tinnes, C., Hohenstein, U., Rössler, W. and Biesdorf, A.
Tackling Model Drifts in Industrial Model-driven Software Product Lines by Means of a Graph Database.
DOI: 10.5220/0011319800003269
In *Proceedings of the 11th International Conference on Data Science, Technology and Applications (DATA 2022)*, pages 146-157
ISBN: 978-989-758-583-8; ISSN: 2184-285X

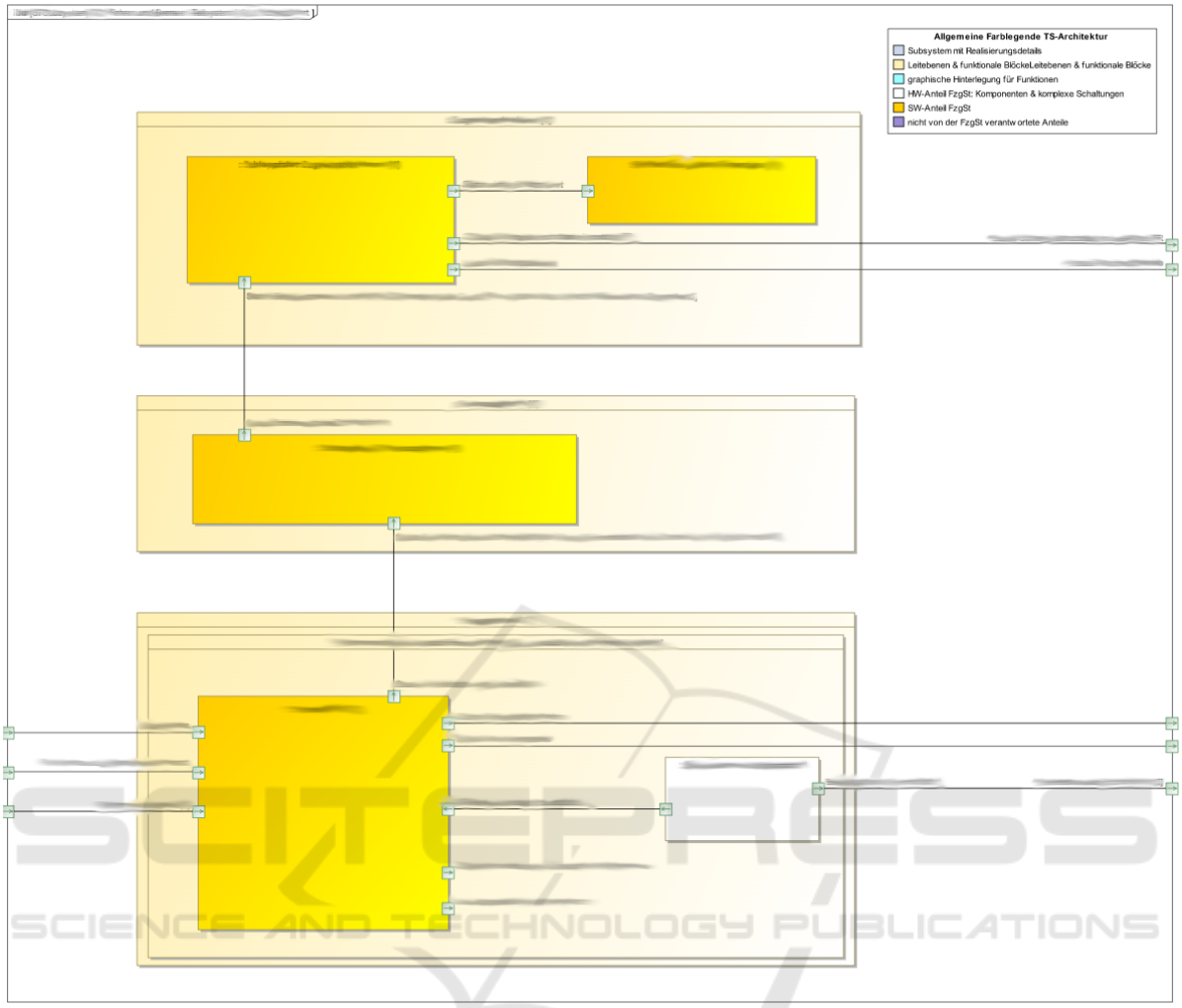


Figure 1: An SysML architectural model artefact in MAGICDRAW(names are obfuscated for reasons of secrecy).

propagation, domain analysis and reuse are negatively affected by causing high efforts and costs.This observation conforms to experiences in the literature (Krüger and Berger, 2020; Rubin et al., 2012; Dubinsky et al., 2013). Due to the high complexity and size of the architectural models, an increasingly large manual effort is currently required for maintaining reuse of the baseline. One root cause for the efforts is the MAGICDRAW difference tool showing a large number of differences, especially (too) many fine-grained differences. Being not handled properly, there is a risk that the efforts even exceed the advantages of reusing commonalities.

To tackle the problem, we use the graph database NEO4J to support an analysis of large model differences. We find that, without any further sophisticated tooling, such a lightweight approach can answer many important questions about model differences and provides a good starting point for further analysis. Particularly, we are enabled to detect and reduce the huge number of model differences.

Section 2 introduces the real-world industrial case study that we conducted, illustrating the problem of comparing large SysML architectural models due to a cloning approach. The preferred solution relies on the graph database NEO4J. Section 3 briefly introduces NEO4J and presents the approach in more detail, whereupon Section 4 presents a few queries sufficient for comparing models. The approach is evaluated in Section 5 by discussing the results that we achieved. Furthermore, we prove the feasibility of the approach by applying it to the modeling data in this project. As a result, the approach turned out to ease the detection of critical model differences and to support central software product-line engineering activities. Section 6 compares our work with existing approaches before Sections 7 concludes the paper with some future ideas.

## 2 PROBLEM STATEMENT AND CASE STUDY

The problem we are tackling in this paper occurred in a concrete industrial project in our company using SysML in MAGICDRAW. The company builds trains, covering small commuter rail to high-speed trains. Trains consist of a lot of software components that implement common functionality such as heating, ventilation, and air conditioning, but also highly safety-relevant features such as the drive control system. A couple of years ago, the software engineering team decided to follow a model-driven engineering (MDE) methodology. Since the code bases have become very large and complex, an MDE approach has been set up to generate parts of the software components' source code from specified SysML models. Moreover, the documentation of the train software, which is required by official authorities, could be generated. Indeed, the train domain is highly regulated and the software is subject to qualification and certification requirements such as IEC61508 (IEC, 2010). It is thus necessary to maintain requirements traceability and exhaustive documentation. Furthermore, traceability between requirements and software components becomes easily manageable; the language SCL itself used for coding the software does not have object-oriented reuse concepts such as inheritance. The MDE approach also enables generating parts of the source code and thus reusing source code components.

About 200 engineers are developing and maintaining the architectural train models. The tool MAGICDRAW is used as the overall modelling environment, since other alternative tools such as ECLIPSE PAPYRUS did not provide the usability necessary for such a large industrial application or failed to scale to the size of the architecture models. To handle the complexity and to keep MAGICDRAW responsive, the model for the entire train software is divided into more than 100 submodels, each covering a particular aspect of functionality, such as the drive and break control.

Having many software components to be reused across different train types leads to a software product line in a MAGICDRAW-proprietary repository, where each train type is kept in a different branch (clone-and-own (Rubin et al., 2013; Dubinsky et al., 2013)). The main line covers the common platform with those parts that are independent of a particular train type. The platform evolves in an *extractive software product line adoption path* (Apel et al., 2013) by (manually) identifying and regularly merging common features into the platform for reuse; changes to common parts in the platform model are propagated to the train types that use the modified parts.

The clone-and-own approach is reasonable in the project compared to a platform-based approach (Krüger and Berger, 2020; Dubinsky et al., 2013; Rubin et al., 2013), since it provides more flexibility in developing new products. Furthermore, an independence of products avoids unintended side effects (Dubinsky et al., 2013; Krüger and Berger, 2020). Other approaches such as feature traces (Fischer et al., 2016; Kehrer et al., 2021) and accompanying tools turned out to be insufficiently mature for industrial environments unfortunately.

However, a drawback is that important tasks such as change propagation, domain analysis, and quality assurance have become time-consuming. For instance, a thorough quality assurance of safety-relevant changes is indispensable to decrease the risk of failing software qualification which in turn would lead to delays in the time-to-market. All those tasks involve the analysis of all the differences between two derivations or between the platform model and a derivation, for example, to check for certain violations, other quality issues, and to identify reuse opportunities during domain analysis. It is thus important to identify a drift in the models, to find causes for the drift, and to take appropriate countermeasures. This is currently a manual and very time-consuming check.

MAGICDRAW indeed has a built-in visual "diff" functionality but exposes about 34,000 differences for just one single of 100 submodels. Despite being displayed in diagrams, some differences are not immediately graphically visible. One has to click on a graphical icon and dive into the textual specification to detect differences at a deeper level. This is far too much to get manually checked and requires more than six days. In particular, it is hard to check whether a deviation is critical or not; this is some tacit knowledge of the engineer who performs the check. For example, some graphical differences are not important like a box being moved in a diagram. Moreover, a diff in MAGICDRAW consumes an excessive amount of RAM due to the size of models and takes a long time in computation (30 minutes for one submodel). This all turned a manual check be time-consuming and error-prone as well, with a strong negative impact.

The consequences of overseeing a critical deviation are manifold. There is a risk of failing a software qualification by official authorities. Hence, the product delivery could be delayed, presumably leading to multi-million Euro penalties. Moreover, every missed critical change can lead to inconsistencies between the common platform and other derived platforms, known as a *model/architecture drift* or *unintentional divergence* (Schmorleiz and Lämmel, 2014; Kehrer et al., 2021). This can be a risk for the entire reuse approach.

Table 1: Requirements.

| | |
|---|---|
| R1 | All deviations must be detected. However, the overwhelming amount of existing differences – which MAGICDRAW exposes – should be reduced to semantic differences, i.e., those that are relevant for the engineers. For example, if an interface has been added to a component, then this fact is sufficient instead of displaying everything that belongs to that interface (what should be considered as a mere consequence). |
| R2 | The comparison results must be easily comprehensible, e.g., in a tabular manner with all relevant information such as ID, component name etc. |
| R3 | The approach should be easy to handle without deeper understanding of the mechanism. |
| R4 | The approach should offer flexibility to scope the investigation, e.g., detecting a typo in an element's name is important, but should be excludable for further analysis. Similarly, tacit knowledge should be addable. |
| R5 | For developing checks, an interactive analysis by means of ad-hoc checks must be possible to try them out. A graphical user interface is reasonable for investigating the model data in depth and diving into details. |

Consequently, any disallowed – and potentially undetected – deviation could cause severe problems which let become an automated check important.

This work focuses on quality assurance tasks. Here, an engineer analyzes the differences between different trainset types or between the common platform and a trainset type in this task in order to identify disallowed deviations. Table 1 summarizes the requirements we collected from engineers to achieve a feasible approach for better supporting the analysis of differences.

# 3 NEO4J APPROACH

Since MAGICDRAW's visual diff tool exposes an overwhelming amount of fine-grained differences, we searched for alternatives to return only relevant modifications and making the check easier and more straightforward.

## 3.1 Generic Extraction Step

To this end, we implemented a proprietary extractor that extracts elements using MAGICDRAW's OpenAPI interface. OpenAPI gives access to the DSL and model elements in a programmatic manner, e.g., for Java.

Our extraction follows a generic approach that is independent of a particular DSL. We derive all the important model parts: The containment hierarchy, which directly corresponds to the containment views in MAGICDRAW, the `typeOf` relationship between DSL elements, attributes of elements such as documentation or arbitrary properties, and connectors between elements.

The derived structure follows a generic data model that reflects the extracted parts. A type `MDBaseElement` contains the general specification with the MAGICDRAW-internal ID, name, level (in the hierarchy), DSL type or stereotype (for example, `Package` or `S7ProxyPort`). If an element is a type of another `MDBaseElement`, a `typeId` refers to the ID of that element. `childElements` contains an array of child nodes, embedded in the same manner, to reflect the containment tree. `MDBaseElement` possesses several subtypes:

- `MDElement` represents an element of the DSL;
- `MDConnection` stores a connection between `MDElement`s;
- `MDDiagram` denotes a graphical diagram;
- `MDPresentationElement` is a graphical component within a diagram.

Listing 1 shows an excerpt of one element being exported in JSON format.

```
{ "elementImplType": "MDElement",
  "id": "a560294_1498144258284_393608_15767",
  "name": "Maintenance operator",
  "dslType" : "Actor",
  "level": 2,
  "typeId": null,
  "belongsTo": null, // if graphical
  "documentation": "Operator of maintenance
  ↪facilities",
  "properties": [ ... ],
  "connections": [ {
    "elementImplType": "MDConnection",
    "direction": "SourceToTarget",
    "id": "a560294_1498144259753_261476_15814",
    "dslType": "Generalization",
    "level": 3,
    "sourceId":
    ↪"a560294_1498144258284_393608_15767",
    "targetId":
    ↪"_a560294_1498144258300_317304_15770"
    ...
  } ],
  "childElements":
    [ { "elementImplType" : "MDElement",
        ... same structure for child ...
      }, ...
    ]
}
```

Listing 1: JSON excerpt of derived information.

There is a clear separation between the DSL elements and their graphical representations in diagrams or tables, keeping the link between elements and their use in graphical representations by means of a

belongsTo relationship. A MDDiagram possesses x/y coordinates, width, height, and a reference to the MDElement it represents. Indeed, an MDElement might possess several graphical representations in different diagrams.

An MDBaseElement might possess connections, which have a direction, a certain dslType, a sourceId (which is the node itself) and a targetId referring to the related element due to technical UML reasons.

Further properties of MDElement's are kept as key/-value pairs with a type MDProperty.

Since the data model is independent of a concrete DSL (although DSL names are available as values of dslType), the data extraction is applicable for any DSL.

To compare two models, these models must be processed as described before.

### 3.2 Transfer to Neo4j

To gain more flexibility in analyzing the models, the JSON outcome is transferred into the graph database NEO4J.

The NEO4J data model consists of nodes and relationships. *Nodes* possess properties and are mainly used to represent entities. A *relationship* connects two nodes and is guaranteed to have a valid source and target node. Relationships are the key feature of graph databases as they allow for finding related data. A *traversal* is a typical way to query a graph, navigating from a node to related nodes by following relationships. Despite having a direction, relationships are traversable in either direction. Both nodes and relationships may have properties. *Properties* are named values where the name (or key) is a string. A *label* in NEO4J can assign types to nodes and relationships; all nodes with the same label belong to the same logical set. A node may be labelled with any number of labels, even none. NEO4J queries can work with these sets, making queries easier to understand and more efficient to execute.

The mapping of the generic data model to NEO4J is as follows. All MDBaseElements are mapped to nodes, obtaining two labels according to the class in the generic model (e.g., MDElement or MDDiagram) and the DSL type (e.g., S7ProxyPort). MDProperty's are stored as nodes with a label Property, referring to their MDBaseElement by means of a relationship. The specific types of references (containment, connections, typeOf etc.) are also reflected by relationships. References, which have been stored by IDs in JSON, are resolved by explicit relationships between elements.

To distinguish two models, a root node of type MDProject is introduced for each loaded model: Each node uniquely refers to its model root by means of a belongsTo relationship.

An important property is that MAGICDRAW identifiers are not changing when taking the platform and deriving a derivate. Thus, identical elements in two imported models, have the same unique MAGICDRAW ID; we call them *pendants*. The query in Listing 2 sets up another relationship refersTo between those pendants based upon IDs:

```
MATCH
(np)-[:belongsTo]->(rp:MDProject{id:"Platform"}),
(na)-[:belongsTo]->(ra:MDProject{id:"ProjectA"})
WHERE  np.id=na.id
CREATE (np)-[:refersTo]->(na)
```

Listing 2: CQL query let refer all elements in *Platform* to elements in *ProjectA*.

The query is formulated in NEO4J's Cypher query language (CQL) in a declarative manner. MATCH identifies nodes and relationships. (...) represents a node and -[...]- a relationship between two nodes; those relationships might ask for a certain direction -[]->. Labels can be placed behind a colon. Hence, (np) represents a node of any type (as the label is left out), whereas (rp:MDProject) is a node with label, i.e., of type MDProject. (np) must be related to a MDProject node (rp:MDProject) via a relationship of type belongsTo. (rp) has to satisfy the condition {id:"Platform"}. Similarly, (na) must belong to "ProjectA" (i.e., (ra)). CREATE establishes a relationship of type refersTo between both if they possess the same ID (np.id=na.id).

## 4 COMPARISON QUERIES

Having the data in NEO4J enables us to query for both the generic and the DSL elements with CQL. Hence, calculating the difference between two models can be performed by a collection of CQL queries. Figure 2 gives an example for a visual query result in NEO4J comparing a node (ID=a560294_1506950258974_794259_327249, name='ATP Fahrtrichtung') in two projects, "Platform" and "ProjectA". The different types of relationships become visible: the newly created refersTo and belongsTo as well as childOf (reflecting the containment hierarchy), hasType, hasProperty, and connection. Colors indicate labels. Selecting a particular node, its name and labels become visible (e.g., key and value of a green Property node).

Please note the more readable representation of connections. There is a direct relationship between two nodes, and the direction (e.g., source_to_target),

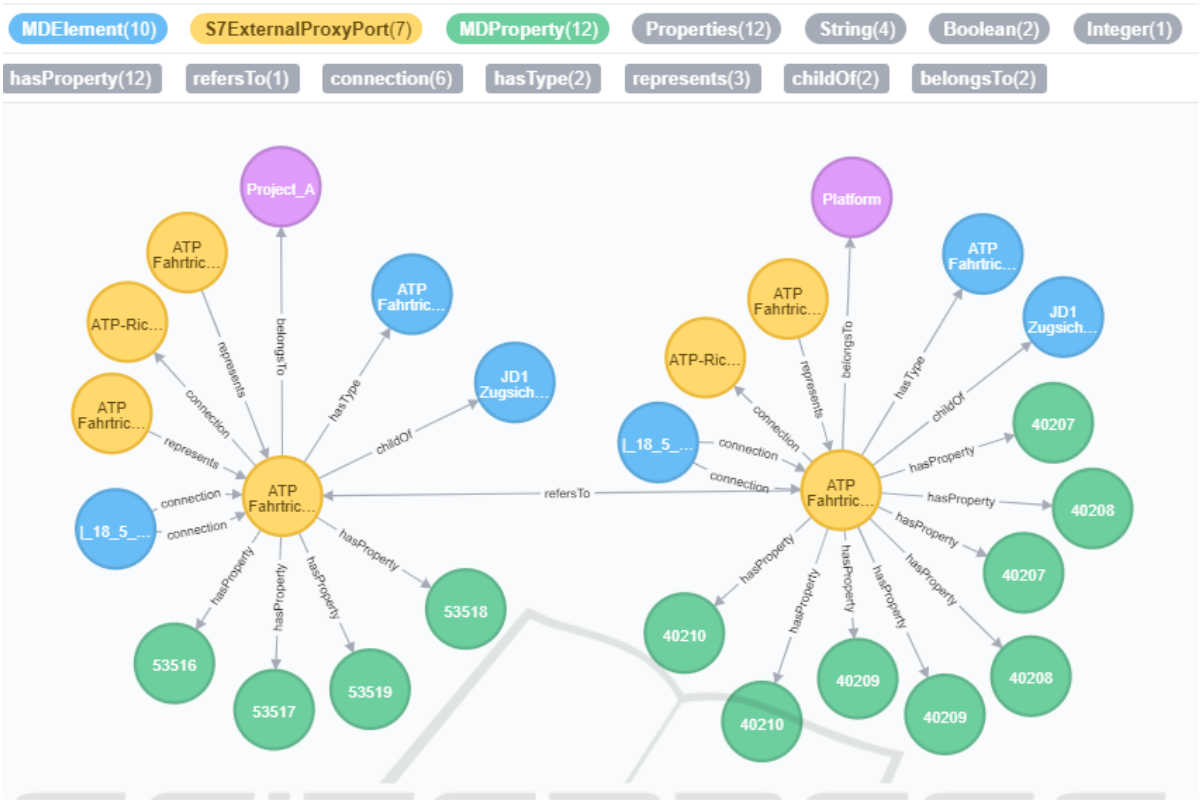


Figure 2: Two nodes to be compared and their relationships.

`dslType`, and `MDConnection ID` are attached to the relationship.

The main goal of this paper is now to find a better way to determine the real, i.e., "semantic", differences (which should be much less than MAGICDRAW's 34,000). Thereby, we want to evaluate the power of NEO4J's CQL for comparing huge models wrt. expressiveness and performance and to determine the number of required queries. To this end, we want to investigate whether we are able to implement some generic, i.e., DSL agnostic, checks that are sufficient to cover all the deviations.

As a first result, it turned out that the queries listed in Table 2 are sufficient to detect all the relevant differences. In the following, we discuss the corresponding CQL queries.

```
MATCH
(np)-[:belongsTo]->(rp:MDProject{id:"Platform"}),
(na)-[:belongsTo]->(ra:MDProject{id:"ProjectA"}),
(np)-[:refersTo]->(na)
WHERE np.id = na.id AND np.name <> na.name
AND NOT 'Properties' IN labels(np)
RETURN np.id, np.name as Name1, na.name as Name2
```

Listing 3: Different names for pendants (Query 1).

The query for Q1 in Listing 3 searches for nodes `(na)` and `(np)` of any label that belong to the projects to be compared (via the `belongsTo` relationship), where both nodes should possess the same ID, but different names; the `refersTo` relationship is redundant to the condition `np.id=na.id`, but useful to visualize this relationship. Nodes of label `Properties` are explicitly excluded since handled in Q4. Figure 3 shows the resulting table, displaying the name differences in columns `Name1` and `Name2`, quite often some typos. In addition to ID and names, the labels (especially the DSL types) can also be issued in the `RETURN` clause.

Table 2: Comparison Queries.

| | |
|---|---|
| Q1 | Which pendants have different names? |
| Q2 | Which pendants have additional or missing sons? |
| Q3 | Which pendants have different parent nodes? |
| Q4 | Which pendants possess different MAGICDRAW properties (wrt. `hasProperty`)? Cardinalities are specified here, which affect the generated software. |
| Q5 | Which pendants possess a different documentation (i.e., `doc(umentation)`)? |
| Q6 | Which pendants refer to different types, i.e., labels? |
| Q7 | Which pendants have different connected elements (via `connection`)? |

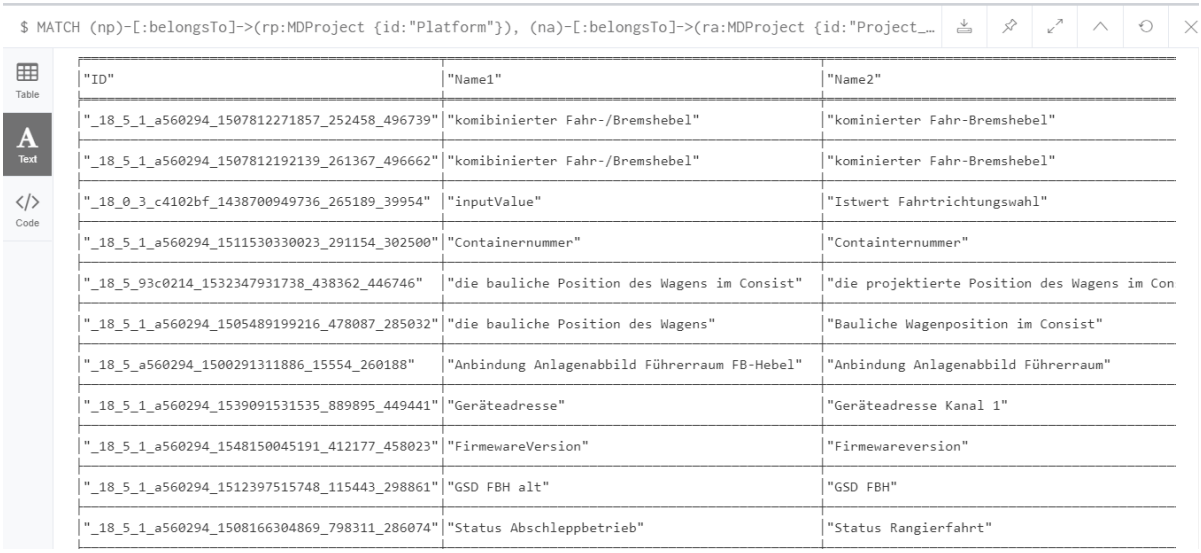


Figure 3: Displaying different names.

Especially labels make the decision easier to check whether the deviation is allowed or not. Hence, it is quite easy to tell critical components apart from non-critical ones. Furthermore, non-critical components can be excluded from the result by adding further filters, e.g., on labels. Anyway, it is also possible to obtain a graphical result (similar to Figure 2) by returning nodes and relationships as `np, na, rel` instead of properties.

As indicated, MAGICDRAW's diff tool leads to an excessive number of differences. Determining only the "semantic" differences is in fact one major problem to tackle. For example, if an element has been added, we are not interested in all sub-elements of that element, but the added element itself is sufficient to know. Moreover, a single new element often leads to several modifications to keep internal associations consistent. Listing 4 determines those nodes. Missing sons are handled analogously.

```
MATCH
(sa)-[:belongsTo]->(ra:MDProject{id:"ProjectA"}),
(sa)-[:childOf]->(fa),
(fp)-[:belongsTo]->(rp:MDProject{id:"Platform"})
WHERE fa.id = fp.id AND NOT ()-[:refersTo]->(sa)
AND 'MDElement' IN labels(sa)
RETURN DISTINCT sa.id AS id, sa.name AS name,
  fa.id AS fid, fa.name AS fname,
  fa.level AS flevel, labels(sa) AS sonlabels,
  labels(fa) AS flabels
```

Listing 4: Newly added sons in ProjectA (Query 2).

Son `(sa)` must belong to "ProjectA" and be child of a father `(fa)`. There must be a pendant `(fp)` that `refersTo (fa)` in "Platform" with the same ID. The `WHERE` condition states that there must not exist any pendant for son `(sa)`. Moreover, graphical representations are excluded by a condition `'MDElement' IN labels(sa)`. `RETURN` yields for each missing node its ID and name, the father's ID, name, and level, and the labels of son and father. The number of results can easily be reduced by adding additional filters incrementally. For instance, differences at levels $>= 8$ might be too detailed and could be excluded by adding a subcondition "`and sa.level<8`". Anyway, the tabular result view allows for quickly checking the deviations.

Query Q3 in Listing 5 has a similar structure: `(np)` and `(na)` are the "identical" nodes (`np.id=na.id`), and `(fp)` and `(fa)` their fathers, resp.

```
MATCH
(np)-[:belongsTo]->(rp:MDProject{id:"Platform"}),
(na)-[:belongsTo]->(ra:MDProject{id:"ProjectA"}),
(np)-[:childOf]->(fp),(na)-[:childOf]->(fa)
WHERE np.id=na.id AND fp.id <> fa.id
RETURN np.id, np.level, na.level
```

Listing 5: Identical nodes with different parents (Query 3).

Determining pendants with different properties is more complex and challenged us and NEO4J. However, we could benefit from the fact that queries can be cascaded in CQL. Query Q4 in Listing 6 starts with nodes `np` for "Platform" and determines the properties `prp` of each node, thereby restricting the nodes to `MDElement`s. From the result, we select (`WITH`) the property `prp`, the node ID, and the set of attached labels for successive subqueries. Having ordered the properties by key, i.e., obtaining an ordered list, another `WITH` clause converts property keys and values into a collection (`collect`). Next, the nodes `na` of "ProjectA" and their properties `pra` are handled analogously, thereby referring to `id1`. The corresponding `MATCH` is `OPTIONAL`

in order to handle empty sets properly. Finally, the sorted list of keys and values are compared.

```
MATCH
(np)-[:belongsTo]->(rp:MDProject{id:"Platform"}),
(np)-[:hasProperty]->(prp)
WHERE "MDElement" IN labels(np)
WITH prp, np.id AS id1, labels(np) as label
ORDER BY prp.key
WITH id1, label, collect(prp.key) AS keys1,
  collect(prp.value) AS values1
OPTIONAL MATCH
(na)-[:belongsTo]->(ra:MDProject{id:"ProjectA"}),
(na)-[:hasProperty]->(pra)
WHERE na.id = id1
WITH id1, label, keys1, values1, pr
ORDER BY pra.key
WITH id1, label, keys1, values1,
  collect(pra.key) AS keys2,
  collect(pra.value) AS values2
WHERE keys1 <> keys2 OR values1 <> values2
RETURN DISTINCT id1, label,keys1, values1,
  keys2, values2
```

Listing 6: Check for pendants with different properties (Query 4).

Listing 7 handles Query Q5. The documentation is stored as a node property `doc`, similar to `id` and `name`. The query checks for different `doc` values (`np.doc <> na.doc`) or having a documentation only at one side (e.g., `EXISTS(np.doc) AND NOT EXISTS(na.doc)`).

```
MATCH
(np)-[:belongsTo]->(rp:MDProject{id:"Platform"}),
(na)-[:belongsTo]->(ra:MDProject{id:"ProjectA"}),
(np)-[r:refersTo]->(na)
WHERE np.id=na.id
AND (np.doc <> na.doc
     OR (EXISTS(np.doc) AND NOT EXISTS(na.doc))
     OR (EXISTS(na.doc) AND NOT EXISTS(np.doc)))
RETURN np.id,na.doc,np.doc
```

Listing 7: Check for pendants with different documentation (Query 5 ).

Query Q6 is split into three different subqueries: a node has a type but the pendant has none (Q6a), a node has no type but the pendant does (Q6b), and both have a type, but different ones (Q6c). From a syntactical point of view, this could be expressed in a single query as well, however, performance then suffers drastically.

```
MATCH -- Q6a (Q6b analogously)
(np)-[:belongsTo]->(rp:MDProject{id:"Platform"}),
(na)-[:belongsTo]->(ra:MDProject{id:"ProjectA"}),
(np)-[ht:hasType]->(tp),(np)-[r:refersTo]->(na)
WHERE np.id = na.id AND NOT (na)-[:hasType]->()
RETURN np,na,tp,ht,r
```

Listing 8: Check for pendants with different types (Query 6a/b).

```
MATCH -- Q6c
(np)-[:belongsTo]->(rp:MDProject{id:"Platform"}),
(na)-[:belongsTo]->(ra:MDProject{id:"ProjectA"}),
(np)-[htp:hasType]->(tp),
(na)-[hta:hasType]->(ta),
(np)-[r:refersTo]->(na)
WHERE np.id = na.id AND tp.id <> ta.id
RETURN DISTINCT np.id,na.id,tp.id,ta.id
```

Listing 9: Check for pendants with different types (Query 6c).

Query Q7 is handled similarly by three subqueries, but not shown here.

So far, this a quite general consideration of differences, which already reduces the number of differences to semantic ones heavily. However, as already mentioned, some deviations are allowed in certain contexts. To handle this aspect, special conditions can be added to better reflect the analyzer's knowledge. For example, some model elements are marked as *sealed*, which means they must not be changed neither in their external interfaces nor their internals. Hence, the following query returns all those elements with such a property valued "true" with all their sons `s` recursively thanks to '*' behind `childOf`. The result can then be checked for any differences.

```
MATCH
(np)-[:belongsTo]->(rp:MDProject{id:"Platform"}),
(np)-[h:hasProperty]->(p),
(s)-[c:childOf*]->(np)
WHERE p.key='sealed' AND p.value="true"
RETURN np.id, s.id
```

Listing 10: Further scoping (sealed components).

## 5 EVALUATION

We start our evaluation with judging the expressiveness and performance of NEO4J. It turns out that NEO4J's CQL is a very powerful query mechanism. We were able to express all the necessary checks (as listed in Table 2) and could not detect any limitations in computation. We were surprised that even very elaborated checks, e.g., Listing 6, can be specified due the possibility to cascade queries, although it took us some time to find a solution. Moreover, performance is absolutely acceptable. Indeed, we applied query tuning sometimes. For instance, an explicit comparison of IDs in addition to the equivalent `belongsTo` relationship speeds up queries. And in Q6 (9), we calculated the nodes having a type in one project but none in the other, and to compare differences only for those nodes with a type in several queries.

In summary, the NEO4J-based approach brought up several advantages wrt. the requirements in Table 1:

**R1**: Concerning the overall approach, we checked the output of our NEO4J-based analysis carefully with a domain expert who usually controls the differences in a manual manner. First, everything we detected by means of CQL queries was correct. Even more, several critical problems, which were not recognized during the manual check could be discovered, e.g., typos in names (cf. Figure 3). The other way round, all real differences seemed to be detected within our analysis. Honestly, we did not check all the violations, but picked up a large number of relevant deviations.

It is possible to reduce the model diff to seven CQL queries – achieving much less, now semantic differences. The results of all the checks are summarized in Table 3. For example, moving a subtree to another parent node leads to 11763 node differences in MAGICDRAW compared to 60 violations (cf. query Q3).

Indeed, the total number of nodes is less than the sum of detected incidents in Table 3 since the query results contain overlaps in nodes: Some nodes have, e.g., additional children, but also further properties and/or connections. Anyway, the numbers expose the number of violations in each check while the number of distinct nodes is less relevant.

**R2**: A tabular text representation of query results (cf. Figure 3) helps a lot to browse through and to judge the findings. It turned out to be useful to return the labels and the level, too. Then it becomes easier to decide whether a certain difference is critical or not, e.g., because it occurs at a deeper level or is not a critical component. This means that an experienced engineer is able to quickly further reduce the occurrences, e.g., by adding `NOT 'MDDiagram' IN labels(n)` to exclude graphical representations. Similarly, an ad-hoc investigation and more advanced scoping (e.g., restricting the scope to `SoftwareComponentInstances` only) by asking for specific labels can be easily added without recompilation.

**R3**: With executing only seven predefined queries, many of the analysis tasks could be simplified. Because of the graphical and tabular representation, results can be understood without deeper understanding of graph databases, the querying language, or metamodel specific aspects.

**R4**: As shown in Section 4, there is enough flexibility to scope the search and to adjust queries according to tacit engineers' knowledge, e.g., Listing 10. For example, there exists some slight kind of feature model in the DSL, however, without any dedicated tool support. Thus, a specific feature allows for new sons (for that feature) while having removed some others that are actually replaced by the feature. Those more complex conditions can be formulated in CQL as well.

Table 3: Comparison Numbers.

| | Name | Occurences |
|---|---|---|
| Q1 | different names | 744 nodes |
| Q2 | additional sons | 947 parents |
| | missing sons | 980 parents |
| Q3 | nodes with different parents | 60 nodes |
| Q4 | no properties in one project | 53 nodes |
| | different properties | 3840 nodes |
| Q5 | different documentation | 91 occ's |
| Q6 | pendant has no type | 12 nodes |
| | pendant has additional type | 112 nodes |
| | different types | 38 nodes |
| Q7 | no connections at one side | 717 occ's |
| | different connections | 345 occ's |

**R5**: There are huge time savings compared to an implementation based upon comparing JSON data in a programming language, especially in the testing phase when queries are developed incrementally. Queries can be adjusted and run immediately without any program/compile/check cycle. Moreover, the visualization helps to understand query results and the DSL "implementation": Diving into DSL internals is possible by checking an element (given by an ID) and all its surroundings, e.g., `MATCH (n {id:"_18_5_1_a560294_1558807659_964294"}), (n)-[r]-(x) RETURN n,x,r` which, in fact, returns the result in Figure 2. It becomes immediately visible that the "Platform" node possesses more properties (`hasProperty`). This is especially useful to check the correctness of CQL queries.

# 6 RELATED WORK

Models are first-class artifacts in MDE, and as such subject to permanent changes. Several studies in the literature support and manage the evolution of models by means of collaborative work on models, versioning of models, or quality assurance of models (Khalil and Dingel, 2013; Arendt and Taentzer, 2013; Paige et al., 2016; Kehrer, 2015; Van Der Straeten et al., 2009). The importance of quality assurance in model evolution or MDE in general has often been discussed. For example, (Krogstie et al., 2006) and (Mohagheghi and Dehlen, 2008) suggest generic quality frameworks for MDE, while(Arendt and Taentzer, 2013) tackle model smell detection and resolution. Another approach by (Giraldo et al., 2014) integrates the metaphor of technical debt into MDE. All the proposed solutions are built on top of the Eclipse Modeling Framework (EMF) and work on the level of models and not on model differences. Our approach is thought to handle large model

differences and support engineers in understanding large model differences.

While model evolution focuses on the differences in the temporal dimension, other work also handles the product dimension in a product line, especially the industrial challenges of variability management. For instance, (Chen and Babar, 2010) present a survey that discusses organizational challenges, complexity and visualizing complex variability, extraction of variability from technical artifacts, evolution of variability itself, variability modeling and documentation, design decision management, and knowledge management and other tool support. (Berger et al., 2020) explicitly mention MDE as a challenge for variability management and the lack of SPLE concepts in many domain specific languages. Some work attempts to unify version and variability management by proposing *variation control systems*. A classification of variation control systems is given by (Linsbauer et al., 2017).

Another approach to support the comparison of two derivations is to condense the model differences. Another approach (Tinnes et al., 2021) aggregates multiple fine-grained differences in larger patterns based on semantic lifting (Kehrer et al., 2011) and model transformation mining (Tinnes et al., 2021). This allows filtering model differences for specific change patterns, too. Our approach is more flexible. Queries can easily be adjusted or new queries can be formulated to answer specific questions, whereas in the approach (Tinnes et al., 2021) new change patterns have to be defined. Furthermore, our approach is independent of the tooling stack.

Closely related to our approach, several query languages for models have been introduced in the past two decades, mainly for the Eclipse Modeling Framework and UML-based tools such as the Object Constraint Language (OCL) (OCL, 2014). The EMF ecosystem brought up many model query languages and tools such as EMF Query (EMF, ), FunnyQT (Horn, 2013), and Emf-IncQuery (Bergmann et al., 2011).Unfortunately, there is no formal comparison of the power and expressiveness of those query languages. Some deficits, mostly for OCL, could not be detected for NEO4J.

In general, the NEO4J approach does not depend on other frameworks like EMF and is therefore easy to employ. Furthermore, queries are easy to understand and can flexibly be adjusted due to the graph-based syntax. Additionally, NEO4J is an industry proven tool and scalable to large amounts of modeling data.

Concerning the use of NEO4J, several papers exist focusing on performance, especially compared to relational databases. Most NEO4J comparisons use artificial test scenarios, for instance, (Vicknair et al., 2010) who experiments with different artificial data sets of 1,000, 5,000, 10,000 and 100,000 nodes with payload attributes.

Other comparisons use data sets coming from real applications. (Khan et al., 2017) compare Oracle 11g and the Neo4j graph databases 5 (3.0.3 community edition) using a proprietary Medical Diagnostic System. The data set comprises of about 28,000 patients, 625,721 patient visits, 869,666 patient_IssueMed records, to mention the main tables. Five count queries join two or three tables. Their result show that Neo4j performs much better when the data set size increases. While Oracle performs queries in a few seconds (depending on the query), Neo4j requires about 0.3 sec.

(Joishi and Sureka, 2015) use some process-mining algorithms for their comparison of MySQL and Neo4j : Similar-Task (finding similarity between actors based on the intersection of activities) and Sub-Contract (causal dependencies between actors in carrying out a business process). MySQL is 32 times faster than Neo4j for the similarity calculation. Concerning subcontracts, Neo4j attains a performance boost of a magnitude of 7x over MySQL.

But due to our knowledge, no research has been applied in the context of comparing huge architectural models thereby judging the capabilities and benefits of NEO4J.

## 7 CONCLUSION

In this paper, we introduced an industrial case study where the tool MAGICDRAW is used in a model-driven engineering manner. System architectures of a product family are modeled using an extended SysML DSL to support reuse of architecture models. In this context, one major issue arises: To effectively compare two large MAGICDRAW models. Indeed, MAGICDRAW has tool support to display differences, however, the result consists of ten-thousands of differences which are hardly to comprehend. To handle this issue, we presented a flexible approach based on the NEO4J graph database. More precisely, comparisons are formulated and detected by means of CQL queries.

As a major result, the number of differences could be drastically reduced to “semantic” differences, which better aggregate differences in the sense that adding a new element does not display all subelements. We also achieve flexibility to let engineers scope the context in order to further reduce the differences by tacit knowledge. The power of NEO4J queries turned out to be sufficient, even complex queries could be formulated. The overall perfor-

mance is acceptable although some tuning was partially required.

The approach can be generalized to other tools, too, due to a central data model. In case of other tools than MAGICDRAW, a different type of analyzer is required to feed the data model. For instance, it should be no problem to apply the approach to EMF-based modelling tools. Similarly, other graph databases can be applied by implementing a different transfer from the data model. Whether the query functionality and performance is sufficient certainly depends on the tool.

Our future work intends to improve the current pipeline. The steps to derive the content of MAGICDRAW models, to transfer the data to NEO4J, where validation is performed, can be reduced to one step. Moreover, the pipeline should be invokable within MAGICDRAW. In this context, we also have to tackle the only remaining performance issue: Without any further improvement, the import steps take a few hours, while the validation itself – query evaluation – is pretty fast.

## REFERENCES


Eclipse modeling framework (emf) model query. https://www.eclipse.org/emf-query/. Last access: April 4, 2022.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer.

Arendt, T. and Taentzer, G. (2013). A tool environment for quality assurance based on the Eclipse Modeling Framework. In *ASE*, pages 141–184. IEEE/ACM.

Batoryayer, A. (2004). Scaling step-wise refinement. *TSE*, 30(6):355–371.

Berger, T., Steghöfer, J.-P., Ziadi, T., Robin, J., and Martinez, J. (2020). The state of adoption and the challenges of systematic variability management in industry. *ESE*.

Bergmann, G., Ujhelyi, Z., Ráth, I., and Varró, D. (2011). A graph query language for EMF models. In *LNCS*, volume 6707 LNCS, pages 167–182.

Chen, L. and Babar, M. (2010). Variability management in software product lines: An investigation of contemporary industrial challenges. In *LNCS*, volume 6287 LNCS, pages 166–180.

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., and Czarnecki, K. (2013). An exploratory study of cloning in industrial software product lines. In *CSMR*, pages 25–34. IEEE.

Fischer, S., Linsbauer, L., Lopez-Herrejon, R., and Egyed, A. (2016). A vision for enhancing clone-and-own with systematic reuse for developing software variants. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, P252:95–96.

Giraldo, F., España, S., Pineda, M., Giraldo, W., and Pastor, O. (2014). Integrating technical debt into MDE. *CEUR Workshop Proceedings*, 1164:145–152.

Horn, T. (2013). Model querying with funnyqt. In Duddy, K. and Kappel, G., editors, *Theory and Practice of Model Transformations*, pages 56–57, Berlin, Heidelberg. Springer.

IEC (2010). 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. Technical report, IEC.

Joishi, J. and Sureka, A. (2015). Graph or relational databases: A speed comparison for process mining algorithm. *Proc. of 19th International Database Engineering &; Applications Symposium, Yokohama*.

Kehrer, T. (2015). *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.

Kehrer, T., Kelter, U., and Taentzer, G. (2011). A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *ASE*, pages 163–172. ACM/IEEE.

Kehrer, T., Thüm, T., Schultheiß, A., and Bittner, P. (2021). Bridging the gap between clone-and-own and software product lines. In *ICSE*, pages 21–25. IEEE/ACM.

Khalil, A. and Dingel, J. (2013). Supporting the evolution of UML models in model driven software development: a survey. *School of Computing, Queen's University, Ontario*.

Khan, W., Ahmed, E., and Shahzad, W. (2017). Predictive performance comparison analysis of relational & nosql graph databases. *Int. Journal of Advanced Computer Science and Applications 8(5), January 2017*.

Krogstie, J., Sindre, G., and Jørgensen, H. (2006). Process models representing knowledge for action: A revised quality framework. *European Journal of Information Systems*, 15(1):91–102.

Krueger, C. (1993). Software Reuse. *ObjectWorld Conference*.

Krüger, J. and Berger, T. (2020). An empirical analysis of the costs of clone-and platform-oriented software reuse. In *ESEC/FSE*, pages 432–444.

Linsbauer, L., Berger, T., and Grünbacher, P. (2017). A classification of variation control systems. *ACM SIGPLAN Notices*, 52(12):49–62.

Mohagheghi, P. and Dehlen, V. (2008). Developing a quality framework for model-driven engineering. In *LNCS*, volume 5002 LNCS, pages 275–286.

No-Magic-Inc. (2021). MagicDraw website. https://www.magicdraw.com/. Last access: April 4, 2022.

OCL (2014). Object Constraint Language, Version 2.4. Technical Report March.

Paige, R., Matragkas, N., and Rose, L. (2016). Evolving models in Model-Driven Engineering: State-of-the-art and future challenges. *JSS*, 111:272–280.

Pohl, K., Böckle, G., and Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*.

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

Rubin, J., Czarnecki, K., and Chechik, M. (2013). Managing cloned variants: A framework and experience. In *SPLC*, pages 101–110. ACM.

Rubin, J., Kirshin, A., Botterweck, G., and Chechik, M. (2012). Managing forked product variants. In *SPLC*, volume 1, pages 156–160.

Schmorleiz, T. and Lämmel, R. (2014). Similarity management via history annotation. *SATToSE 2014—Preproceedings*, page 45.

Tinnes, C., Kehrer, T., Joblin, M., Hohenstein, U., Biesdorf, A., and Apel, S. (2021). Learning domain-specific edit operations from model repositories with fsm. In *ASE*. ACM/IEEE.

Van Der Straeten, R., Mens, T., and Van Baelen, S. (2009). Challenges in model-driven software engineering. In *LNCS*, volume 5421, pages 35–47.

Vicknair, C., Macias, M., Nan. X., Zhao, Z., and et al. (2010). A comparison between a graph and a relational database: A data provenance view. *Proc. of 48th Annual Southeast Regional Conference, 2010, Oxford, (USA)*.