Mining Frequent Structures in Conceptual Models

Mattia Fumagalli[1], Tiago Prince Sales[2], Pedro Paulo F. Barcelos[2], Giovanni Micale[3], Philipp-Lorenz Glaser[5], Dominik Bork[5], Vadim Zaytsev[4], Diego Calvanese[1], Giancarlo Guizzardi[2]

[1]KRDB Research Centre on Knowledge and Data, Free University of Bozen-Bolzano, Bolzano, Italy.
[2]Semantics, Cybersecurity & Services (SCS), University of Twente, Enschede, The Netherlands.
[3]Department of Clinical and Experimental Medicine, University of Catania, Catania, Italy.
[4]Formal Methods and Tools (FMT), University of Twente, Enschede, The Netherlands.
[5]TU Wien, Business Informatics Group, Vienna, Austria.

Abstract

The challenge of using structured methods to represent knowledge is a well-documented issue in conceptual modeling and has been the focus of extensive research. It is widely recognized that adopting *modeling patterns* offers an effective structural approach for designing conceptual models. Patterns, in this context, refer to generalizable, recurring structures that provide solutions to common design problems. They significantly enhance both the *understanding* and *improvement* of the modeling process. Numerous experimental studies have demonstrated the undeniable value of using patterns in conceptual modeling. Despite this, the task of identifying patterns in conceptual models remains highly complex, and there is currently no systematic method for pattern discovery. To address this gap, this paper proposes a general approach for *discovering frequent structures* in conceptual modeling languages as a means to support pattern identification. Specifically, we focus on uncovering recurring structures that reflect the usage patterns of a given conceptual modeling language. As proof of concept, we implement our approach by focusing on two widely-used conceptual modeling languages. This implementation includes an exploratory tool that integrates a *frequent subgraph mining algorithm* with *graph manipulation techniques.* The tool processes multiple conceptual models and identifies recurrent structures based on various criteria. We validate the tool using two state-of-the-art curated datasets: one consisting of models encoded in OntoUML and the other in ArchiMate. The primary objective of our approach is to provide a support tool for language engineers. This tool can be used to identify both effective and ineffective modeling practices, enabling the refinement and evolution of conceptual modeling languages. Furthermore, it facilitates the reuse of accumulated expertise, ultimately supporting the creation of higher-quality models in a given language.

Keywords: Conceptual Modeling, Mining Conceptual Models, Frequent Subgraph Mining, Recurrent Modeling Structures, Modeling Patterns

1 Introduction

Conceptual modeling is a highly complex task. As empirical results show [1–5], this is often due to multiple factors, such as the cognitive limitation of modelers, the lack of information about the domain to be modeled, and the nature of the conceptual modeling language being adopted. For this reason, one of the main concerns of language engineers is overcoming modelers' difficulties and devising conceptual modeling languages that assure as much as possible the high-level quality of the output conceptual model. Patterns play a key role in this regard. These recurrent structures are *generalizable* solutions to design problems that help in *understanding* and *improving* the process of creating models. Specifically, patterns represent examples of good (or bad) recurring practices that language engineers can identify by looking at the practical application of the language itself. By discovering and adopting recurrent structures, language engineers can then evolve the modeling language itself, speeding up and facilitating its application and easing the reuse of working experiences, by also helping to avoid common errors or misconceptions.

During the past decade, modeling patterns have been widely adopted by language designers [6, 7] and, consequently, discovering them from design experiences has become of paramount importance. Still, the task of discovering recurring modeling structures across conceptual models presents multiple challenges. For instance, consider that *.i* recurrent modeling practices can only be discovered by observing a vast number of conceptual modeling examples, and *.ii* the assessment of the conceptual models and the identification of recurrent modeling structures often require analysis activities that are highly time-consuming when performed manually, as seen in processes like *modularization*, *frequency calculation*, or *constructs correlation*.

The analysis of conceptual patterns and the discovery of recurring structures or useful patterns has emerged as a distinct research path that has seen significant growth in recent years. This research, as evidenced by seminal works like [6–9], focuses on analyzing reference patterns—used as case studies—to infer modeling strategies for specific problems. More recent work has explored the application of automated techniques to support heuristic tasks traditionally performed manually [10–13].

In this article, we join the ongoing research efforts to develop automated support for facilitating the empirical discovery of recurring modeling structures. Our contribution is an approach that integrates several state-of-the-art techniques, designed as an exploratory tool to enable users to interactively discover and analyze frequent structures. The implementation of this approach is based on gSpan [14], a widely recognized *frequent subgraph mining algorithm*. Through a command-line interface, this tool enables users to:

.i select the models to be processed,

.ii prepare the data to be mined by focusing on specific information and filtering out concepts that might not be relevant for that type of analysis,

.iii select certain features that the output frequent structures should have (e.g., number of nodes, frequency, or dissimilarities from known patterns),

.iv visualize the output frequent structures and query the input models to verify the domain occurrences of that structure, and

.v cluster the output structures to simplify the user's final assessment.

The main scope of our solution is to offer a support facility for language engineers that aims at exploiting bad/good practices to evolve and maintain the conceptual modeling language but also to favor the reuse of encoded experience in designing better models with the given language.

We evaluate our approach using two large and curated state-of-the-art datasets of frequently used conceptual modeling languages, in particular, the OntoUML dataset [15] and the ArchiMate dataset [16]. We adopted datasets encoded in OntoUML and ArchiMate languages for several reasons. Primarily, these languages are based on pre-defined patterns, which are frequently used across different models to address common modeling issues. This allows us to evaluate our approach to finding useful structures that are already known by the designers of each language. Second, for our evaluations, we can access curated high-quality datasets, encoded in a uniform format, thus improving our findings' comprehensibility and reliability. Third, for our evaluations and experiment design, we can obtain feedback directly from the authors of the languages,[1] showing how this may help enabling us to gather valuable information about the practical utility of the approach. Finally, OntoUML and ArchiMate have a variety of constructs and features that allow us to assume that if our approach works for these languages, it also works for widely used standard languages, such as UML or BPMN.

The outline of this paper is as follows: Section 2 explores the concept of "pattern" in conceptual modeling, focusing particularly on "recurrent modeling structures". This section also provides a brief overview of the primary technique employed for automating the mining process. Section 3 enumerates the requirements that guided the design of our methodology. In Section 4, we introduce our framework, elaborating on its workflow and individual components. Section 5 details the implementation of our approach. In Section 6 and Section 7, we discuss the experiments and demonstrations conducted to validate our solution. In Section 8, we analyze insights gained from the evaluation and demonstration processes. Section 9 situates our work within the context of existing literature. Finally, Section 10 offers reflections on our findings.

2 Research Baseline

2.1 Patterns in Conceptual Modeling

Currently, our proposal primarily focuses on patterns in conceptual models. These model fragments represent recurrent structures formed by modeling constructs from a specific conceptual modeling language. Keeping track of these patterns allows one to understand how a language is applied, thus offering powerful means to language designers for maintaining and evolving the language itself. Using recurrent structures,

[1] In particular, we leveraged this feedback in the OntoUML scenario.

one can, e.g., discover the most common combinations of language constructs, identify language dialects for specific application domains, verify possible restrictions in using constructs or their combinations, and determine the frequent *subversions* of the language. The notion of *systematic language subversion* [17] refers to an ungrammatical use of a language's constructs that becomes recurrent in a language community signalling a design limitation of that language. It is closely related to *coding traditions* [18] that cover coding policies, notational guidelines, naming conventions, implementation patterns, programming idioms, etc.

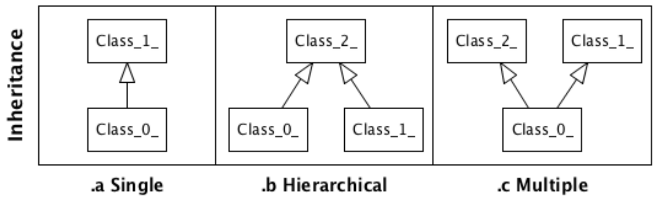


Fig. 1: Example of inheritance recurrent structures in UML class diagrams [19].

Figure 1 shows examples of common usages of the generalization relationship in UML class diagrams. These structures represent the ways by which modelers represent intentional sub-typing (inheritance) between classes (see *.a*, *.b* and *.c* types), and can be taken as useful insights for understanding the language application. For instance, given a set of models about a certain domain, it is possible to discover that the *multiple inheritance* (*.c*) is used to capture class compositionally [19]. This observation might trigger language designers to suggest class composition relations when modelers use two or more generalization relations for a class.

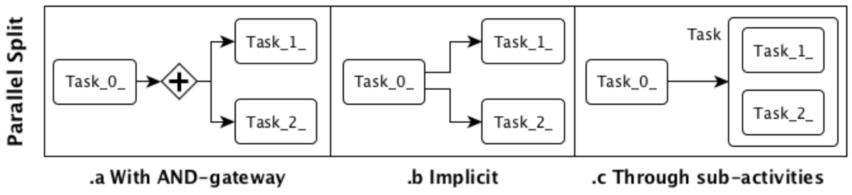


Fig. 2: Example of parallel split recurrent structures in BPMN diagrams [20].

Similarly, Figure 2 represents three examples of recurring structures in BPMN models. These patterns correspond to equivalent ways of modeling tasks parallel splitting (see *.a*, *.b*, and *.c* types) and are related to how people design control flows. Just as with UML structural patterns, these BPMN fragments can offer interesting insights into the language's usage. For instance, by observing multiple occurrences of the implicit parallel split (.b), language designers may decide to force modelers to make explicit the gateway for the splitting, to avoid unintended instantiations of the model (i.e., where the splitting may be intended as an ``OR'', instead of an ``AND'', depending on the modelers' scope).

2.2 Frequent Subgraph Mining

Frequent Subgraph Mining (FSM) is a well-known technique [21] used to find frequent subgraphs in a graph dataset. This technique typically involves a dual-phase approach. The initial stage involves the *creation of subgraph candidates* [21], where subgraphs in the graph dataset are searched and proposed for the analysis, while the subsequent step entails *evaluating the frequency* of the generated subgraphs to ascertain their prevalence.

FSM presents two distinct problem formulations: *.i graph transaction-based FSM*, and *.ii single graph-based FSM*. In graph transaction-based FSM, the input dataset comprises a collection of medium-sized graphs termed "transactions" [22]. On the other hand, single graph-based FSM uses a single, significantly large graph as input. In this setting, a subgraph, denoted as g, is deemed frequent if its occurrence count surpasses a predefined threshold value, commonly referred to as the *support threshold*. The support calculation varies according to the problem formulation. In a transactional dataset, the support $\alpha (0 < \alpha < 1)$ of a graph is the ratio of the number of transactions to which this graph occurs to the total number of transactions. In a single large graph, the support of a graph is the number of its occurrences in this graph.

Formally, given a graph $G = (V_g , E_g)$, where V_g is a set of (possibly labeled) vertices and E_g is a set of (possibly labeled) edges, then a graph $H = (V_h, E_h)$ is a subgraph of G if and only if its vertices and edges are a subset of the vertices ($V_h \subseteq V_g$) and edges ($E_h \subseteq E_g$) of graph G, and the vertex subset include all endpoints of the edge subset. A subgraph is considered frequent when its support equals or exceeds the user-defined minimum support threshold.

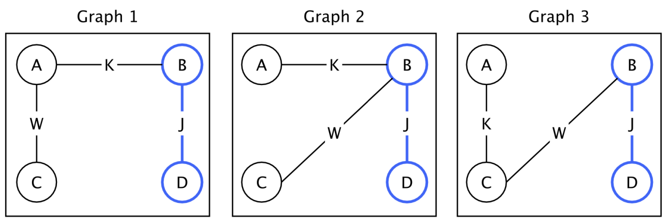


Fig. 3: Example of three graphs, with subgraph $B - J - D$ having *support (frequency)* of 3

Let us consider the three graphs represented in Figure 3. Suppose that we want to discover all the (connected) subgraphs with *at least two nodes* that occur in *at least three graphs*. By adopting *minimum support 3* as a parameter and applying an FSM algorithm, we can obtain the set of all subgraphs appearing in at least three graphs, namely $B - J - D$.

In practical scenarios, different tools or algorithms can identify frequent subgraphs. In this paper, we adopt gSpan [14], a popular state-of-the-art solution. This algorithm is often used for discovering frequent subgraphs in large graph databases and has variants that work on conceptual graphs [23], typed linked data [24], chemical compound data [25], large disk-based databases [26], data with differential privacy [27], etc. It uses an approach based on a *minimum DFS code* [28] that allows the efficient generation of candidate subgraphs and pruning of infrequent ones. The algorithm iteratively discovers all frequent subgraphs containing an edge and then expands the search to include larger subgraphs. The database is shrunk as the search continues, and only graphs containing the current subgraph are considered. Studies show that gSpan can mine large frequent subgraphs with lower frequency and high performance [21].

3 Requirements

Our design for the proposed approach is based on an initial *problem identification* activity. During this phase, we gathered feedback from five potential users of our approach: expert language engineers, who have been involved in the development of different conceptual modeling languages and who have been working on the identification of conceptual modeling patterns. We performed open-ended interviews and the main open questions we asked were:

.i *What is the relevance of an approach for facilitating the empirical discovery of structural modeling patterns in conceptual models?*, and

.ii *What is required in order to facilitate the empirical discovery of modeling patterns?*

This preliminary step helped us improve our awareness of the problem, better understanding the related work, and better identifying the features that our solution should offer to the end-users. The feedback from experts was crucial in defining both functional and non-functional requirements, which are needed to design the approach and evaluate the artifact in which our contribution is embedded.

We mapped the key features that specify *what* our approach should do into the following *functional requirements*:

R1. *Interestingness:* The approach should facilitate the discovery of *subjectively interesting* patterns, namely recurrent structures that can be considered interesting according to the user's interpretation. Here, the notion of "subjectively interesting" is inspired by the work from Silberschatz and Tuzhilin [29], where a pattern is ranked as interesting by a user mainly because *.i* it is considered *exploitable* for the modeling activities, or *.ii* it contradicts some users' expectations.

R2. *Customization:* The approach should allow engineers to manipulate the input models to obtain different patterns, which may vary in size (number of nodes or edges), may be related to different constructs (see, for instance, *taxonomical* vs. *non-taxonomical* structures), or maybe taken at different levels of granularity (e.g., in UML *compositions* and *aggregations* relations may be taken just as *associations*). The approach should also allow the user to filter the output of the discovery process according to custom parameters.

R3. *Comprehension:* The approach should assist engineers in the process of assessing and analyzing the output structures. This should be feasible by accounting for multiple types of frequency, e.g., *relative frequency*, as the number of occurrences of a structure in each model; *total frequency*, as the overall occurrences for a given pattern; *frequency across models*, as the number of models in which a pattern occurs. Moreover, this should be supported by a human-readable visualization format of the output structures.

We mapped the key features that specify *how* the approach should perform its functions into the following *non-functional requirements*:

R4. *Performance.* The approach should outperform the human discovery activity in terms of time. Moreover, the conceptual models processing and the mining step should *happen* in a reasonable amount of time, even with a *large* amount of data, where "reasonable" and "large" are considered concerning related work [30, 31] (e.g., hundreds of models).

R5. *Compatibility.* The approach should be able to support pattern discovery in multiple conceptual modeling languages.

4 Method

Our approach is represented as a workflow, consisting of several tasks, which can be categorized into three main phases: *preparation*, *discovery*, and *assessment.* These phases allow the user to intervene in the workflow whose inputs, outputs, and dependencies are combined as from Figure 4 below.

4.1 Preparing the Input Data

The "Preparation" phase focuses on transforming the input conceptual models in a format that is processable by the subsequent steps. In this phase, we have two main tasks. First, the *Importing (0)* task consists of taking a set of conceptual models M encoded in a language (e.g., UML or BPMN) and transforming each model $m_i \in M$ into a *Labeled Property Graph (LPG)* g_j. Such a task is denoted by a white box in Figure 4 because it is language-dependent, namely it requires an *ad hoc* transformation for each source conceptual modeling language taken into consideration. At first glance, this might seem straightforward since conceptual models are essentially graphs. That, however, is not always the case. Consider, for instance, the transformation of UML class diagrams into graphs. The simple solution is transforming classes into nodes and generalizations and associations into edges. Still, if we want to convert, generalization sets, association classes, generalizations between associations, cardinalities, and several other constructs, this solution no longer works. Moreover, a challenge for the available frequent subgraph mining approaches is accounting for *complex (semantically-rich)* graph data [21, 32, 33]. There are several algorithms each of which admits graphs with different characteristics (e.g., labeled vs. unlabeled or directed vs. undirected).

In the conceptual modeling context, we encounter construct-rich languages that challenge the capabilities of existing FSM solutions. For instance, some languages support multiple labels for classes and relations, multiple relationships between the same

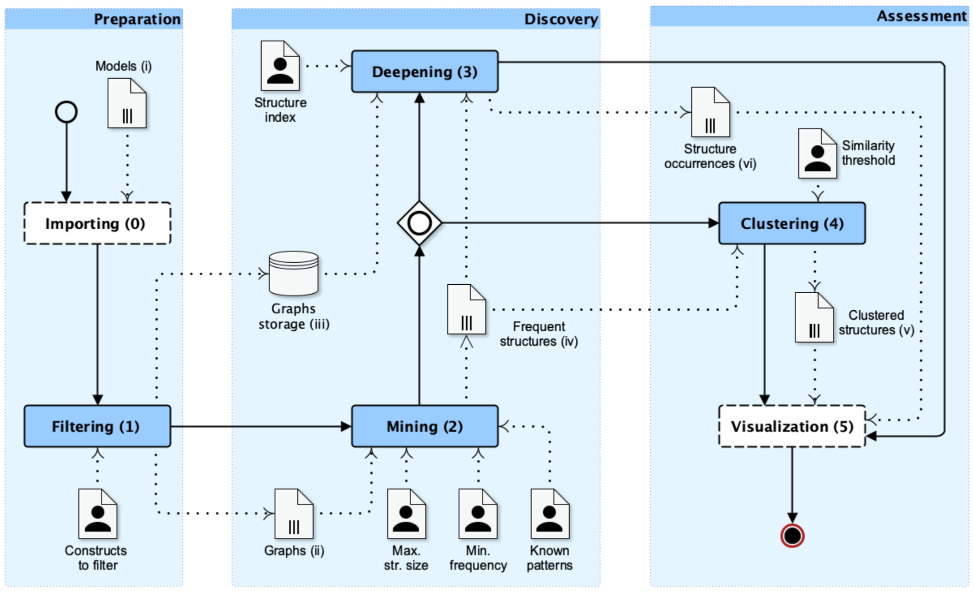


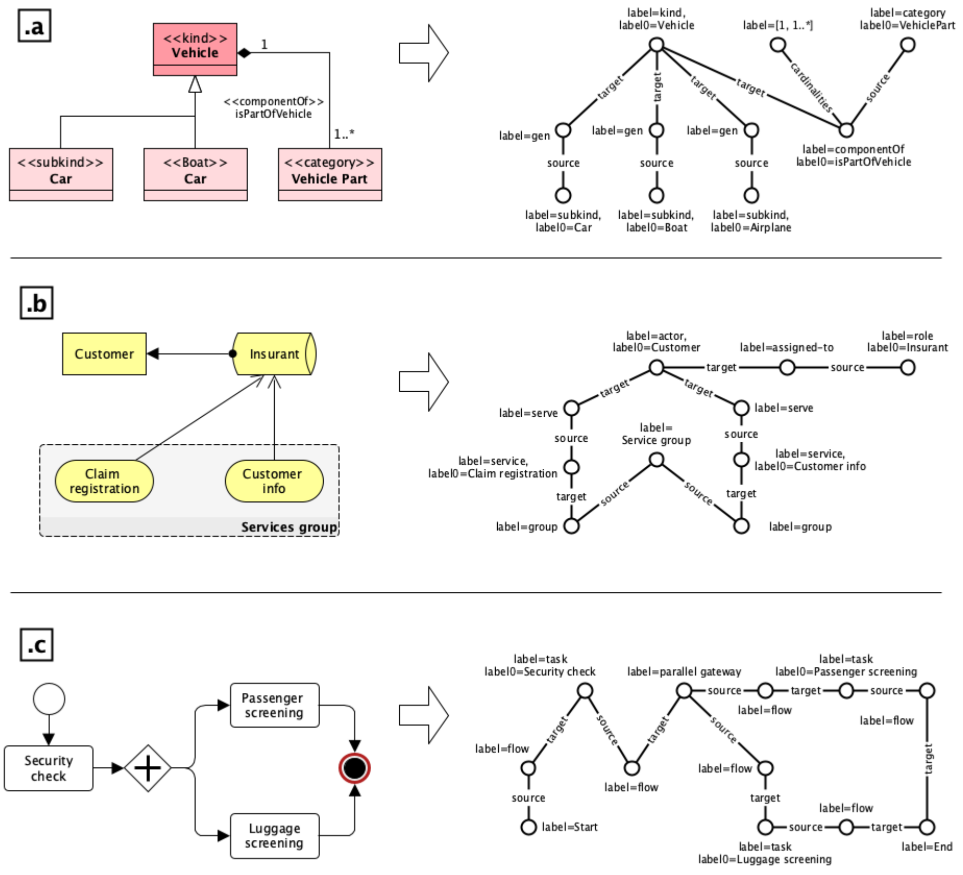
Fig. 4: The frequent structures discovery workflow.

classes, and even classes of relations. This requires the mining algorithm to potentially handle various types of graphs, such as multi-labeled, directed, or multi-graphs. To the best of our knowledge, no existing algorithm fully accommodates all these graph types. Therefore, we ensured full compatibility with all algorithms by addressing the problem at its source. The importing process generates an LPG that captures all the information from conceptual models encoded in any conceptual modeling language, ensuring the output format is always suitable for the mining algorithm.

Algorithm 1: Description of the generic importing algorithm.

```
Data: Set of Conceptual Models $M$
Result: Set of Labeled Property Graphs (LPG) $G$
for each conceptual model $m \in M$ do
    for each concept $c$ in $m$ do
        map $c$ to a node $n_c$;
        if $c$ represents a relation then
            Connect $n_c$ to the source node with an edge $e_s$ labeled
             "source";
            Connect $n_c$ to the target node with an edge $e_t$ labeled
             "target";
        end
        for each property $p$ of $c$ do
            assign a label $l_p$ to node $n_c$;
        end
    end
    create an undirected labeled graph $g$ using the set of nodes and
     edges;
    add $g$ to $G$;
end
```

Algorithm 1 presents a simplified overview of the importing process. The core idea is to *reify* every element in the input conceptual model. For example, associations, cardinalities, and classes are all mapped into graph nodes with multiple labels. Figure 5 illustrates this transformation process, applied to three models encoded in different modeling languages: *(a)* OntoUML, *(b)* ArchiMate, and *(c)* BPMN. On the right, the resulting LBGs are shown. The edge labels "source" and "target" are used to preserve the directionality of the relationships.



.a
<<kind>> Vehicle
1
<<componentOf>> isPartOfVehicle
1..*
<<subkind>> Car
<<Boat>> Car
<<category>> Vehicle Part
label=kind, label0=Vehicle
label=[1, 1..*]
label=category label0=VehiclePart
target
cardinalities
source
label=gen
label=componentOf label0=isPartOfVehicle
label=subkind, label0=Car
label=subkind, label0=Boat
label=subkind, label0=Airplane
.b
Customer
Insurant
Claim registration
Customer info
Services group
label=actor, label0=Customer
label=assigned-to
label=role label0=Insurant
label=serve
label= Service group
label=service, label0=Claim registration
label=service, label0=Customer info
label=group
.c
Security check
Passenger screening
Luggage screening
label=task label0=Security check
label=parallel gateway
label=task label0=Passenger screening
label=flow
label=Start
label=End
label=task label0=Luggage screening

Fig. 5: Example of transformation from conceptual models to labeled property graphs.

In the *Filtering (1)* task, users can choose which language constructs to exclude from the models. For instance, with OntoUML, one may want to seek patterns only involving classes decorated with certain stereotypes, or involving only classes, generalizations, and generalization sets. Finally, the outputs of the filtering task are a set of (possibly filtered) LPGs (*.ii* in Figure 4) to be given as input to the mining algorithm and the storage of all the generated graphs (*.iii*, which can be used to run the *Deepening (3)* task (see Section 4.2 below).

4.2 Discovering Structures Across and Within Models

The "Discovery" phase is the workflow's core and aims to generate the candidate patterns in a format that can then be processed and made accessible to the user for the final assessment. This phase, in turn, is composed of two main tasks.

Mining (2), the first task, involves applying an FSM algorithm. This allows for an interaction with the user, who can select:

.i the *minimum structure size* for the output patterns (e.g., filter out patterns that have less than five nodes),

.ii the *minimum frequency* threshold for the output patterns (e.g., filter out patterns that occur less than 30 times across models), and

.iii the *known patterns* to be excluded from the final output (e.g., the user can provide graphs representing known patterns as input. These will be excluded from the final output to maximize the discovery of results that contain new information).

The output of the mining task will then comprise a list of the discovered patterns in a format that eases the final assessment, along with a *pattern index*, a list of indexes of the *source models* in which the pattern occurs, and a *model frequency value* calculated as the number of conceptual models that contain the pattern. For instance, given five models, the model frequency cannot be more than '5'.

An optional task named *Deepening (3)* allows users to select a pattern from the previous task and discover the occurrences of this pattern within each input model. This operation can be run for each of the output patterns and implies using *the selected pattern as a query to be run over the graphs storage generated via the importing task*. The output of the deepening task is a set of pattern occurrences for each selected pattern. This allows the user to derive the *total frequency* for each pattern, calculated as the sum of the pattern occurrences within each input model (e.g., given five models, if the pattern occurs twice in each model the total frequency will be of '10', but the model frequency will be '5'). Through this step, the user can extract information that is related to the specific usage of the pattern within the reference model domain. For instance, via deepening, we can infer that, in a pattern, a node with stereotype `Kind` is associated with the label 'Vehicle' in a model and 'Means of transportation' in another model.

4.3 Assessing the Output Frequent Structures

The "Assessment" phase aids users in analyzing the output. Similar to previous phases, this one comprises two main tasks.

The *Clustering (4)* task groups output structures—specifically, the recurrent or frequent structures from the mining task—based on their similarities.[2] This should

[2]Notice that in the approach we are proposing, to assess whether one conceptual model sub-structure is close to another, we take inspiration from conceptual model similarity techniques [34–36], where the main task is to find similar models given a reference model input, to categorizing models according to their characteristics.

Algorithm 2: Partial Description of the Clustering Algorithm.

```
Data: Set of of Frequent Structures $F$
Result: Set of Clustered Frequent Structures $C$
for every frequent structure $f$, where $f \in F$ do
    extract $\alpha$ as the number of nodes in $f$;
    extract $\beta$ the number of edges in $f$;
    extract the adjacency matrix $A$ for $f$;
    for each node $n$ where $n \in f$ do
        extract the associated labels $L$;
    end
    for $\alpha$, $\beta$, $A$ and $L$ do
        flatten $A$ into a vector $v_A$ concatenating its rows;
        generate a vector $v_f = (\alpha, \beta, v_A, L)$;
        store $v_f$ in a list of vectors $V$;
    end
end
for every vector $v_f^i \in V$ do
    compute similarity with every other vector $v_f^j \in V$;
    take as input a similarity threshold $\gamma$;
    create the set of clusters $C$ according to $\gamma$[a];
end
```

[a]Note that, in our scenario, if we have three patterns, A, B, and C, and, according to a certain threshold, A is similar to B and B is similar to C, this implies that A is also similar to C, and A, B and C are in the same cluster.

facilitate the user's process of scraping and consulting the output. Often, the number of outputs produced can surpass hundreds, and numerous structures can have similar characteristics. For example, some structures might differ by just one label (in one structure we might have three nodes where one is labeled as `Kind` and the other two specialize it as `Subkind`, while in another structure we might have three nodes where one is `Kind`, and the other two specialize it as `Subkind` and `Role`, respectively. In this sense, a cluster of patterns can be considered by the user as a set of variations, which sometimes may refer also to the same structural pattern (e.g., where a kind is specialized by two or three `Subkinds`). To enable clustering, in our process, we extract key characteristics from the graphs that represent each pattern. At present, the steps addressed to enable the clustering step are as from Algorithm 2.

Currently, we adopt a single feature extraction method to calculate structure similarity. However, this does not prevent adopting different feature extraction or embedding techniques. This is valid also for the approach used to assess similarity, which now is limited to a *cosine similarity* [37], but in the future, it can be extended to other measures.

The *Visualization (4)* task is crucial, focusing on creating an output that maximizes users' comprehension and engagement. This process revolves around transforming generated patterns, which inherently encapsulate the reified structure of input graphs, back into their original format — a multi-directed graph. In this graphical representation, relationships, generalizations, and associations manifest as edges, complete with corresponding cardinalities to provide a richer context.

The significance of this visual representation lies in its ability to empower users in several ways. For example, by translating the outputs from the FSM algorithm into

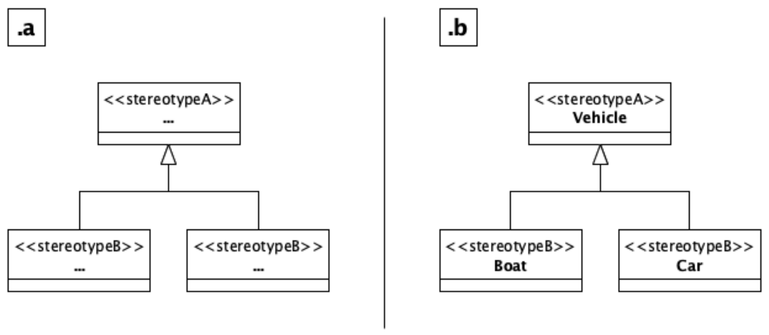


Fig. 6: Visualization example of recurrent structure *(.a)* and a related occurrence *(.b)*.

multi-directed graphs, users can better grasp complex relationships and hierarchies within the data. This enables deeper insights and understanding. Figure 6 exemplifies the transformation power of visualization. On the left-hand side, you can observe a recurrent structure. On the right, you are presented with one of its occurrences, generated through the deepening task. This visual juxtaposition illustrates how the raw outputs of the mining and deepening tasks can be transformed into a visually accessible and actionable representation, empowering users to easily extract valuable knowledge from their data.

5 Implementation

We developed a command-line Python application for our approach, allowing users to interactively configure the process, select the input conceptual modeling language, manipulate the data, and assess the mining algorithm's output. The scripts we created are available at `https://github.com/unibz-core/CM-Mining`.

For tasks involving graph importing, processing, and transformation, we used the *NetworkX library*[3], a powerful Python package for creating, analyzing, and visualizing complex networks. NetworkX played a crucial role in each step of the entire workflow. We integrated the *Grandiso* library[4]—a versatile Python library known for its efficient graph-matching and isomorphism-checking capabilities—in tasks involving graph matching, isomorphism detection, and motif search. For pattern mining, we relied on the *gSpan Python* implementation[5] of the frequent subgraph mining algorithm. This library enabled us to discover recurring subgraph patterns within our data. We achieved the visualization of discovered patterns using *PlantUML*[6], a flexible tool for generating UML and ArchiMate diagrams. PlantUML is seamlessly integrated into our framework, enhancing the interpretability of our results through clear and informative visual representations.

[3] `https://networkx.org/`
[4] `https://pypi.org/project/grandiso/`
[5] `https://pypi.org/project/gspan-mining/`
[6] `https://plantuml.com/`

This suite of tools enabled a robust and efficient framework for graph structure discovery and visualization, ensuring we can efficiently access useful information from the mining outputs.

5.1 OntoUML Miner

As an initial proof of concept for our scientific contribution, we adapted our pipeline to support UML class diagrams, focusing specifically on OntoUML models. These represent the output of a trending paradigm situated at the confluence of conceptual modeling and ontology engineering, namely Ontology-driven conceptual modeling (ODCM). ODCM frequently entails the utilization of fundamental ontologies to steer the formulation of conceptual models, modeling languages, and tools [38]. Within this framework, the OntoUML modeling language [39, 40], has risen to prominence as one of the predominant methodologies [38].

Algorithm 3: Partial Description of the OntoUML Importing Algorithm.

```
Data: Set of of Conceptual Models $M$
Result: Set of Labeled Property Graphs (LPG) $G$
for every conceptual model $m$, where $m \in M$ do
    map each class $c$ into a node $n_c$;
    map each association $a$ into a a node $n_a$;
    map each generalization $g$ into a node $n_g$;
    map association cardinalities $\phi$ into a node $n_\phi$;
    for each generalization node $n_g$ do
        Connect $n_g$ to the parent class-node $n_c$ with an edge “general”;
        Connect $n_g$ to the child class-node $n_c$ with an edge “specific”;
    end
    for each association node $n_a$ do
        Connect $n_a$ to the source class-node $n_c$ with an edge “source”;
        Connect $n_a$ to the target class-node $n_c$ with an edge “target”;
        for each cardinalities node $n_\phi$ do
            Connect $n_\phi$ to the corresponding $n_a$ with an edge
             “cardinalities”;
        end
    end
    for each class $c$ do
        create the corresponding label(s) for node $n_c$;
    end
    for each association $a$ do
        create the corresponding label(s) for node $n_a$;
    end
    create $LPG$ $g$;
    add a graph $g$ to $G$;
end
```

OntoUML extends the UML modeling language. Its meta-model is grounded on UFO (Unified Foundational Ontology) [41], a formal theory based on contributions from Formal Ontology in Philosophy, Philosophical Logic, Cognitive Psychology, and Linguistics. UFO is one of the most used foundational ontologies in conceptual modeling and OntoUML is among the most used languages in ontology-driven conceptual modeling [38]. An example of core ontological distinctions underlying OntoUML is represented by the key categories of *object types* (e.g., `Kind`, `Subkind`, `Roles`, and

RoleMixins), *trope types* (e.g., Relator, Mode) and Relations (FormalRelations, MaterialRelations, and ParthoodRelations).[7]

One of the main goals of OntoUML is to support the conceptual modeling activities by making explicit the semantics behind the modelers' design choices, thus enabling key features of the output conceptual models, such as understandability, interoperability, and reusability.

The purpose of the OntoUML miner is to enable the discovery of recurrent structures within OntoUML models, requiring the ability to handle all OntoUML constructs. To achieve this, the module adapts both the importing and visualization steps of the pipeline, which were identified as language-dependent in Figure 4. The OntoUML-specific importing step is detailed in Algorithm 3, providing a specialized version of Algorithm 1. This partial view of the algorithm shows that the input concepts from the conceptual model correspond to specific OntoUML constructs, many of which also appear in UML. For example, taxonomic relations, associations, and relation properties like cardinalities are represented.[8]

As a final remark on the OntoUML visualization step, this process entails adapting the PlantUML transformation so that output patterns are displayed using OntoUML-like notation.

5.2 ArchiMate Miner

Besides OntoUML, we extended our pipeline to support ArchiMate models, demonstrating the adaptability of our mining approach to different modeling languages. ArchiMate, standardized by the Open Group,[9] is one of the most widely used Enterprise Architecture (EA) modeling language [42]. The ArchiMate framework adopts a layered view of an enterprise, where the core entities of an enterprise are categorized along *layers* (e.g., Business, Application, or Technology) and *aspects* (e.g., Active Structure, Passive Structure, or Behavior).

To enable the discovery of frequent structures within ArchiMate models, we adapted our pipeline to accommodate ArchiMate constructs during the language-dependent importing, filtering, and visualization steps.

The ArchiMate-specific importing step is detailed in Algorithm 4, extending the general approach outlined in Algorithm 1. This step transforms ArchiMate models into Labeled Property Graphs (LPGs) by mapping elements and relationships to nodes and assigning labels that capture their name and type information (e.g., BusinessProcess for an element or Assignment for a relationship). Specialization relationships are treated distinctly, with nodes connected by edges labeled general and specific to ensure that hierarchical structures are distinctly represented and allow for additional filtering.

[7]For an in-depth analysis and characterization of the ontological categories underlying OntoUML, the reader is referred to [39, 40].

[8]Note that the example pseudocode presented covers only a subset of constructs. Information on generalization sets, aggregation, and composition relations is not included. However, the importing step can readily handle these elements by applying the same reification strategy outlined in Algorithm 1 (e.g., a generalization set is represented as a node linked to generalization relation nodes, with properties such as disjoint and complete).

[9]https://pubs.opengroup.org/architecture/archimate3-doc/

The filtering step allows users to refine the models before mining, reducing the graph size and, subsequently, the search space. Users can apply filters based on:

.i element types (e.g., `ApplicationComponent`, `TechnologyService`),
.ii layers or aspects (e.g., focus solely on the Technology Layer or Behavior Aspect),
.iii relationship types (e.g., include only `Realization` or `Access` relationships), and
.iv edge labels (e.g., filtering by `general`/`specific` or `source`/`target` labels).

Finally, similar to the OntoUML implementation, the visualization step was adapted to render ArchiMate patterns using PlantUML, which natively supports ArchiMate diagrams.[10]

Algorithm 4: Partial Description of the ArchiMate-specific Importing Algorithm.

```
Data: Set of of ArchiMate Models $M$
Result: Set of Labeled Property Graphs (LPG) $G$
for every model $m$, where $m \in M$ do
    map each element $e$ into a node $n_e$;
    map each relationship $r$ into a a node $n_r$;
    map each specialization $s$ into a node $n_s$;
    for each specialization node $n_s$ do
        Connect $n_s$ to the parent element-node $n_e$ with an edge “general”;
        Connect $n_s$ to the child element-node $n_e$ with an edge “specific”;
    end
    for each relationship node $n_r$ do
        Connect $n_r$ to the source element-node $n_e$ with an edge “source”;
        Connect $n_r$ to the target element-node $n_e$ with an edge “target”;
    end
    for each element $e$ do
        create the corresponding label(s) for node $n_e$;
    end
    for each relationship $a$ do
        create the corresponding label(s) for node $n_r$;
    end
    create $LPG$ $g$;
    add graph $g$ to $G$;
end
```

6 Experiments

In this section, we test our approach through three experiments, keeping as reference the requirements described in Section 3, and addressing the following research questions:

RQ1 *Can the proposed approach generate structures encoding previously recognized interesting patterns?* This research question is aimed at testing whether the proposed solution can discover pre-identified interesting patterns **(R1)**. This research question is also used to check the role of the customization steps in supporting the discovery process **(R2)** and the level of comprehensibility of the outcome **(R3)**.

[10] https://plantuml.com/archimate-diagram

RQ2 *What are the main parameters affecting the performance of the approach?* Here, we want to identify the characteristics of the input data or parameters having a major influence on the performance **(R4)**.

RQ3 *Is the clustering step accurate in grouping structures?* This research question is mainly concerned with **(R3)**. Here, we want to assess the practical utility of a key component in the presentation of the output.

The data about the experiments and detailed instructions for reproducibility are available at this `GitHub` link: `https://github.com/unibz-core/cmining-approach`.

6.1 Experiment 1: Reliability Test

The primary objective of this experiment is to validate the approach with respect to **RQ1**. Specifically, we aim to determine whether the choices made in the importing task enable the discovery of structures similar to those previously identified by domain experts. To achieve this, we used known patterns for each language as a reference and observed the impact of selected input parameters on the mining task outputs. For full control over the validation, we created ten models, each containing instances of the known patterns. This controlled context and small dataset also allowed us to closely monitor how filtering actions effectively prune results that fall outside the search scope.

6.1.1 Using the OntoUML Dataset

This experiment used an *ad hoc* set of models with some OntoUML patterns we already know as input and we checked how many expected patterns were found.

Data: A dataset comprising 10 models developed by the authors of this paper. These are small models (varying from a minimum of *6 classes* and *5 relations* to a maximum of *10 classes* and *8 relations*)[11] that were created specifically for this experiment, taking inspiration from structures that are present in real models, with the goal of reproducing a controlled number of target patterns. The distribution of patterns per model can be observed in Table 1.

Setup: For validation, we used six common OntoUML patterns that served as "litmus test".[12] These patterns were previously manually identified as useful for building OntoUML models by the designers of the language within multiple example models [7, 44]. We represent the selected patterns in Figure 7.

We executed the application seven times (one without adopting parameters to filter specific concepts and the other times with six different configurations, each to find one of the 6 selected target patterns) and we checked whether the proposed solution could discover the pre-identified interesting patterns (**R1**). Moreover, we tested the role of the customization steps in supporting the discovery process (**R2**) and the level of comprehensibility of the outcome (**R3**). First, we conducted a trial adopting no customization facility from the pipeline. As parameters, we selected 3 (the support of the less frequent patterns, i.e., `Relator`, `Subkind`, and `Phase`) as minimum

[11]The model files are available here: `cmining-approach/tree/main/ontouml/experiment1`

[12]A litmus test is "a critical indicator of future success or failure" A is a litmus test for B if A can be effectively used to measure some property of B [43].

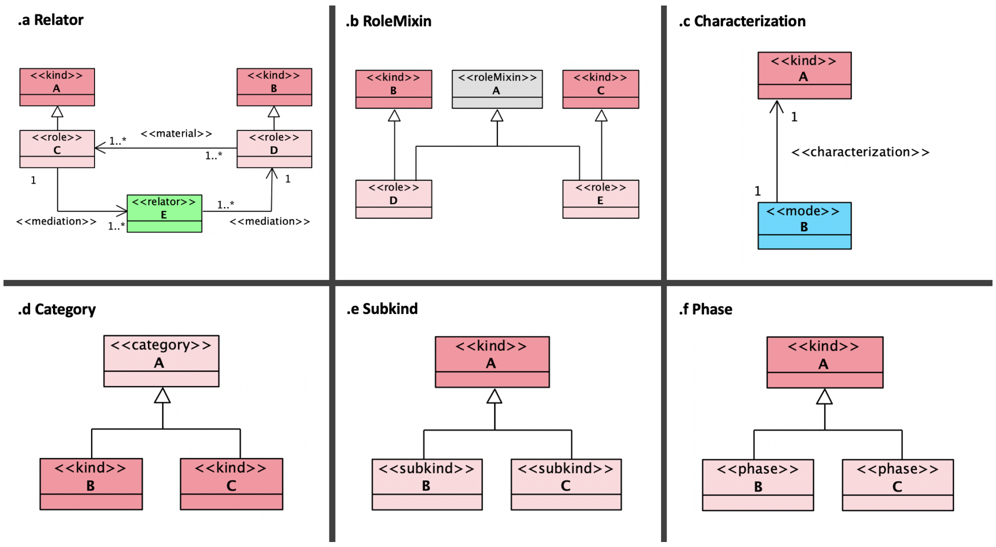


Fig. 7: OntoUML modeling patterns examples [44].

support, and 5 as minimum number of nodes (the size of the smaller pattern, i.e., `Characterization`),[13] so that all patterns can be discovered. For each other trial, we customized the pipeline to find the target pattern, e.g., by filtering some constructs and some other patterns we did not want to discover. For instance, for the `Relator` pattern, we selected `Kind`, `Relator`, and `Role` as target constructs and we filtered out information about the generalizations sets (the customization we adopted in terms of *minimum support*, *nodes number*, and *selected/removed constructs* is reported in Table 2.).[14]

Results: The conducted trial successfully identified all the target patterns, demonstrating the pipeline's proficiency in addressing **R1**. The approach identified all the (classes of) patterns that were explicitly identified *a priori* and correctly counted their occurrences.

The "Neutral" trial yielded an extensive list of output patterns, numbering in the thousands, augmenting the challenge of pinpointing the target pattern.

In subsequent trials, the application of the customization features significantly decreased the number of output patterns, thus showing the key role of the features we

[13]Where we have 2 nodes for the classes, 1 node as an association and 2 nodes for the cardinalities.

[14]To provide context on the OntoUML constructs present in the patterns, here is a brief explanation: A `Kind` is a construct commonly used across models to represent rigid concepts that establish an identity principle. A `Subkind` represents rigid specializations of identity providers, such as `Kinds`, while the `Relator` construct is used to represent *truth-makers* of *material relations—entities* that must exist for two or more individuals to be connected by *material relations*. A `Role` represents anti-rigid specializations of identity providers, like `Kinds`. The `Category` construct serves as a *rigid mixin*, which does not depend on a specific identity principle but is used to aggregate essential properties across individuals with different identity principles. A `RoleMixin` is the equivalent of `Role` for types that aggregate instances with different identity principles. The `Phase` stereotype represents anti-rigid subtypes of identity providers, such as `Kinds`, that arise due to changes in intrinsic properties (for example, a person's age). Finally, `Characterization` is a relation that connects a bearer type with its features. More detailed information on OntoUML constructs can be found in [39] and `https://github.com/OntoUML/ontouml-models`.

Table 1: Patterns distribution over the synthetic data set.

Model	Relator	RoleMixin	Charact.	Category	Subkind	Phase
01	0	0	1	1	1	1
02	0	0	2	1	0	1
03	0	0	1	1	0	2
04	1	0	1	0	0	0
05	1	0	1	1	0	0
06	0	1	0	2	0	0
07	0	1	1	0	1	0
08	1	0	1	1	0	0
09	0	1	0	2	0	0
10	0	1	1	0	1	0
Overall Freq.	3	4	9	9	3	4
Model Freq.	3	4	8	7	3	3

Table 2: Discovered patterns. "Neutral" is the trial where no customization has occurred. Each of the other records represents a trial performed to find a target pattern. For instance, "Relator" was performed to find the corresponding pattern and the total patterns found were 42.

trial	freq.	nodes	filter	freq. strs.
`Neutral`	3	5	*none*	4045
`Relator`	3	12	{ `Select: Kind, Role, Relator` } { `Remove: Charact` }	42
`RoleMixin`	4	10	{ `Select: Kind, Role, Rmixin` } { `Remove: Assoc` }	2
`Charact.`	8	4	{ `Select: Kind, Mode` } {`Remove: Gen` }	1[a]
`Category`	6	4	{ `Select: Kind, Category` } { `Remove: Assoc` }	32
`Subkind`	3	4	{ `Select: Kind, Subkind` } { `Remove: Assoc` }	41
`Phase`	3	4	{ `Select: Kind, Phase` } { `Remove: Assoc` }	89

[1]Note that the result "1" does not mean that only one occurrence of that pattern was found, but that exactly one structure corresponding to that pattern was found. This structure, in turn, can occur multiple times, for example, in this case, the frequency was "8", as expected.

implemented in the discovery process (**R2**). For instance, with the `Relator` pattern, we selectively filtered extraneous stereotypes classes and relations out, resulting in a noteworthy reduction of the number of outputs (from 4045 to 42, see Table 2). We provide a visual representation and explain the `Relator` pattern in Figure 8. Particularly, the pipeline accurately and comprehensively identified all constructs originating from the input graphs.

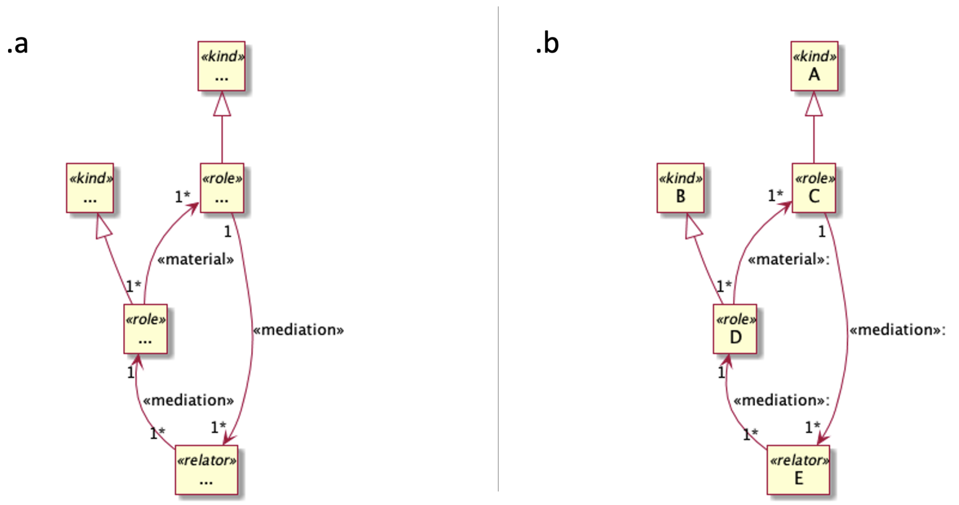


Fig. 8: Example of visualization for a `Relator` pattern occurrence as the pipeline returned it.

6.1.2 Using the ArchiMate Dataset

Similarly to the OntoUML experiment, we used an *ad hoc* set of ArchiMate models, containing various known patterns as input, and we conducted different trials to verify that the expected patterns can be found in the mining output.

Data: This experiment used a synthetic dataset of 10 manually created ArchiMate models, with sizes ranging from 10 to 15 elements and 8 to 12 relationships. Six EA Smells were selected as target anti-patterns for this analysis, as illustrated in Figure 9. These patterns include `Chatty Service` (`CS`), `Combinatorial Explosion` (`CE`), `Cyclic Dependency` (`CD`), `Data Service` (`DS`), `Multifaceted Abstraction` (`MA`), and `Wrong Cuts` (`WC`). EA Smells [45] serve as qualitative indicators of structural inefficiencies and represent potential issues that affect the non-functional aspects of EA models (e.g., maintenance). Analogous to code smells that signal technical debt in source code, EA Smells assesses an organization holistically, beyond purely technical scopes [45]. The EA Smells used in this experiment were selected from an extensive catalog[15] and translated into ArchiMate patterns (cf. [46]) using the provided descriptions. The distribution of the patterns across the dataset is listed in Table 3.

Setup: The experiment was organized into seven trials. The first, referred to as the "neutral" trial, mined patterns without applying any filters to capture all potential patterns. Parameters were set to identify patterns with a frequency of at least 3 (matching the frequency of less frequent patterns, e.g., `Chatty Service` and `Data Service`) and a minimum node count of 5 (corresponding to the size of the smallest patterns, e.g., `Multifaceted Abstraction` and `Wrong Cuts`).

[15] https://swc-public.pages.rwth-aachen.de/smells/ea-smells/

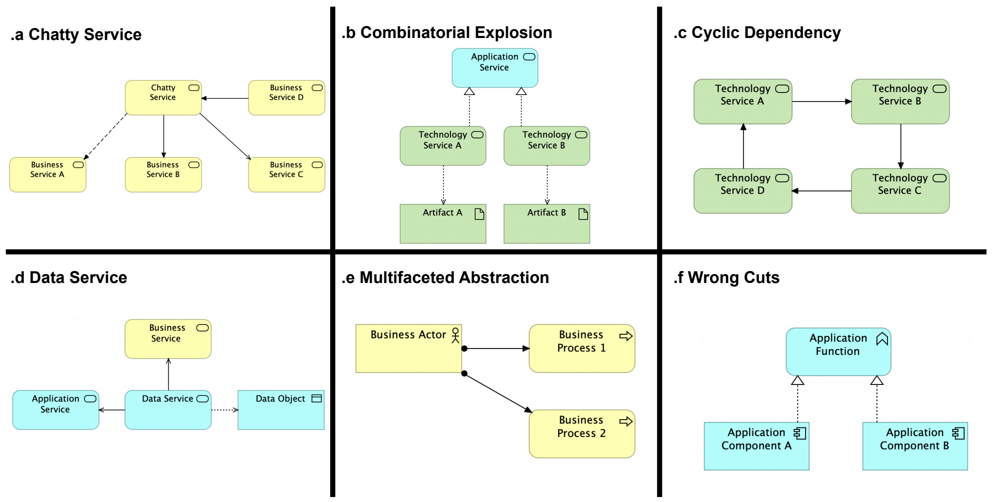


Fig. 9: ArchiMate selected EA Smells Patterns.

Table 3: Patterns distribution over the synthetic ArchiMate dataset

Model	`CS`	`CE`	`CD`	`DS`	`MA`	`WC`
01	1	0	1	0	1	0
02	0	1	1	0	0	2
03	0	1	0	1	0	1
04	1	0	0	1	1	0
05	0	1	0	0	1	2
06	0	0	2	0	0	1
07	0	0	1	0	1	1
08	1	0	1	1	0	0
09	0	1	0	0	2	1
10	0	0	2	0	0	1
Overall Freq.	3	4	8	3	6	9
Model Freq.	3	4	6	3	5	7

The subsequent six trials focused on identifying the individual patterns by applying specific filters to narrow the search space. For instance, `Cyclic Dependency` patterns in the dataset exclusively consist of `TechnologyService` elements and `Triggering` relationships. Hence, filters were applied to include only these elements and relationships while excluding others. The mining parameters, such as the minimum frequency and node count, were also adjusted to ensure patterns of interest were captured. The complete configuration of parameters and filters for each trial is summarized in Table 4.

Results: Table 4 summarizes the results of the seven trials. The neutral trial yielded 80 total patterns, including unrelated or redundant outputs, showcasing the need for customized filtering to refine results. In contrast, the filtered trials successfully identified all target patterns, with each trial isolating the intended structure and significantly reducing the number of output patterns (**R2**). For instance, the `Wrong Cuts` (`WC`) pattern, distributed 9 times across 7 models (see Table 3), was correctly identified as a single recurring pattern during the corresponding trial, when applying the specified parameters and filters. Similarly, all other filtered trials correctly identified the intended EA Smell patterns, confirming the approach's reliability in detecting known structures and accurately counting their occurrences (**R1**). Note that in the `Cyclic Dependency` (`CD`) trial 3 patterns were found, consisting of our target pattern and two duplicates, due to the cycle in the graph structure. An example visualization for a concrete pattern, as returned from the pipeline, is shown in Figure 10.

Table 4: Discovered Patterns

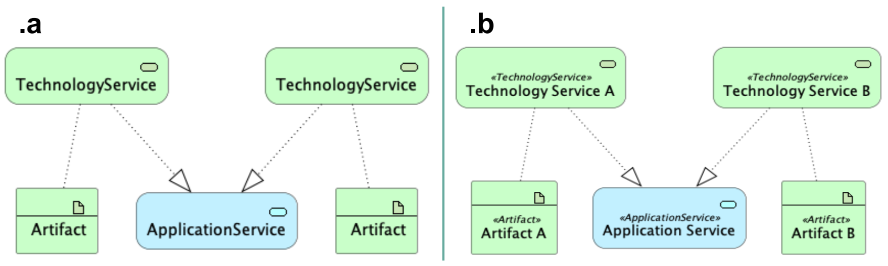
trial	**freq**	**nodes**	**filter**	**freq. strs.**
`Neutral`	3	5	*none*	80
`CS`	3	9	{ `BusinessProcess` } { `Flow`, `Serving`, `Triggering` }	1
`CE`	4	9	{ `TechnologyService`, `Artifact`, `ApplicationService` } { `Realization`, `Access` }	1
`CD`	6	8	{ `TechnologyService` } { `Triggering` }	3
`DS`	3	7	{ `ApplicationService`, `BusinessService`, `DataObject` } { `Access`, `Serving` }	1
`MA`	5	5	{ `BusinessActor`, `BusinessProcess` } { `Assignment` }	1
`WC`	7	5	{ `ApplicationComponent`, `ApplicationFunction` } { `Realization` }	1

6.1.3 Threats to Validity

This experiment focused on a limited set of patterns that, although relatively intricate, represent only a fraction of a broader spectrum of patterns. Moreover, the synthetic datasets used in this study were fairly compact. However, the combination of the patterns we selected from the OntoUML and the ArchiMate scenarios is suitable for assessing whether our approach can discover structures with the required level of expressiveness.

6.2 Experiment 2: Performance Test

This experiment primarily addresses **RQ2**. Here, performance is not measured by the number of patterns identified versus those expected, as the chosen mining algorithm has been verified to be 100% accurate [14], making metrics like precision and recall irrelevant. Likewise, the main focus is not the time required to generate output, given



.a
TechnologyService
TechnologyService
Artifact
ApplicationService
Artifact
.b
«TechnologyService» Technology Service A
«TechnologyService» Technology Service B
«Artifact» Artifact A
«ApplicationService» Application Service
«Artifact» Artifact B

Fig. 10: Example visualization for a `Combinatorial Explosion` pattern and occurrence as the pipeline returned it: **.a** shows the pattern with its related element- and relationship types, while **.b** shows an example occurrence of the pattern including element names.

that FSM tasks are known to be time-consuming. Instead, the goals are: *.i* to assess whether the algorithm can produce results on real datasets containing models of varying sizes when run on basic, affordable hardware, and *.ii* to observe how the execution time scales with an increase in input models.

It is important to note that performance is influenced by the specific implementation choices made for each module in the pipeline, which can vary. For instance, future versions of the application may adopt a different mining algorithm if needed. This highlights the modular nature of our approach, which allows different components to be interchanged to support the same tasks effectively.

6.2.1 Using the OntoUML Dataset

Data: The input data comprised 94 models from a catalog of *OntoUML* models [15]. The catalogue results from a community effort, which collected models of different sizes, describing different domains, and developed for varying purposes in different contexts.

Setup: This validation was executed on a *MacBook Pro* (Retina, 13-inch, Early 2015) with CPU 2,7 GHz Intel Core i5 and was organized in two main trials: a baseline *trial (0)*, where we selected as input the 47 conceptual models from the OntoUML dataset and *trial (1)* where we added 47 more models. The selected models have different sizes, in terms of relations and and classes. As an example, the graphs generated from these sources can have a small number of nodes and relations, namely 47 and 35 respectively (see `lindeberg2022simple-ontorights.json`, in the reference experiments repository), or higher, namely 2.449 and 2.282 (see `indeberg2022full-ontorights.json`). For each step, to increase the amount of data to be handled, we also tested the approach with six different customizations (we show these values in Table 5) concerning partition size parameters and the minimum frequency of the mining task.

Results: Table 5 summarizes the results of this experiment by providing the time taken by the graph partitioning, the mining step, the clustering, and the generation of a visual representation for each pattern. The execution time reveals that the processing time *rises* when:

.i the number of models increases,
.ii the number of nodes for the partitioned graph decreases, and
.iii the minimum frequency threshold is decreased, thus allowing to find a larger number of patterns.

As we might expect, the function that takes the most time is the one dedicated to mining and pattern generation.

Table 5: Experiment 2 results. For each of the two trials (47 and 94 models), we selected the same parameters (nodes and frequency) and provided the number of patterns plus the time taken by four functions of the pipeline.

models	import (s)	nodes	freq.	mining (s)	patterns	clustering (s)	viz. (s)
47	0.7327	5	20	1.375	1	0.011	0.106
		3	20	1.591	15	0.043	1.648
		5	15	3.792	22	0.073	2.257
		3	15	3.950	65	0.422	6.669
		5	10	47.579	246	0.982	16.387
		3	10	48.448	372	1.360	20.387
94	1.238	5	20	130.458	37	0.103	3.781
		3	20	184.239	72	0.503	7.360
		5	15	224.782	141	0.734	9.410
		3	15	229.138	193	0.812	11.500
		5	10	324.781	586	1.990	25.675
		3	10	359.318	662	3.156	31.057

6.2.2 Using the ArchiMate Dataset

Data: For this experiment, we used subsets of 50 and 100 medium-sized ArchiMate models, extracted from the EAModelSet dataset [16]. The EAModelSet is a FAIR dataset comprising over 900 ArchiMate models of varying sizes, collected from GitHub, GenMyModel, and the EA community. From the dataset, we first filtered for English-language models and then selected medium-sized models with 30 to 80 relationships each. The 50-model subset contained 2.862 elements and 3.570 relationships, while the 100-model subset comprised 5.890 elements and 6.920 relationships.

Setup: The experiment was run on the machine adopted also in the experiment with OntoUML models (see Section 6.2.1) Two primary trials were conducted: one with 50 models and the other with 100 models. For each trial, six different parameter combinations were tested, progressively increasing the amount of data to be processed. No filters were applied, meaning all elements and relationships in the models were analyzed. A timeout of 600 seconds (10 minutes) was imposed for the mining step to ensure execution feasibility. The parameter combinations and results are presented in Table 6.

Results: Table 6 summarizes the results, detailing the time taken for each step of the pipeline: graph partitioning (import), mining, clustering, and visualization. For the visualization step, only the time required to generate PlantUML.txt files was measured. As expected, the mining step accounted for most of the execution time. The results show that processing time increased under the following conditions:

.i when the number of total elements and relationships is increased (e.g., moving from 50 to 100 models),

.ii when the minimum node parameter decreased, allowing smaller patterns to be considered,

.iii when the frequency threshold was reduced, resulting in a larger number of patterns being mined.

For the 50-model subset, all parameter combinations were completed successfully within the timeout limit. However, for the 100-model subset, the mining step exceeded the 600-second timeout in both trials with the lowest frequency (10), indicating that processing models of this size with such settings would require more time and more robust hardware. These results reaffirm the high computational cost of the mining step, particularly when handling larger datasets with more lenient parameters. The results also show the need to balance parameter settings with hardware capabilities to ensure efficient execution.

Table 6: Experiment 2 results with the ArchiMate dataset

models	**import (s)**	**nodes**	**freq.**	**mining (s)**	**patterns**	**clustering (s)**	**viz. (s)**
50	0.066	5	20	0.178	1	0.545	0.153
		3	20	0.182	11	0.562	0.137
		5	15	1.257	42	0.571	0.124
		3	15	1.283	79	0.582	0.147
		5	10	66.810	502	0.593	0.159
		3	10	87.120	604	0.624	0.187
100	0.154	5	20	125.447	8	0.657	0.201
		3	20	131.459	38	0.683	0.207
		5	15	455.807	68	1.032	0.314
		3	15	507.853	93	1.154	0.462
		5	10	600+	-	-	-
		3	10	600+	-	-	-

6.2.3 Threats to Validity

The primary challenge to the validity of this experiment stems from the absence of a comparative analysis involving multiple devices. The ultimate performance might be influenced by additional variables not present in the current setting. Nonetheless, the configuration we employed can be viewed as a stress test, given its resemblance to the common features found in widely utilized laptops and the absence of a high-performance CPU.

6.3 Experiment 3: Clustering Accuracy Test

The goal of this final experiment is to address **RQ3** by testing the utility of the clustering component. It is important to note that the clustering component is not the primary focus of our contribution, so this is not the context to compare our implementation choices with existing alternatives (particularly regarding the embedding techniques and similarity computations used). Instead, the purpose of this test is mainly to assess how effectively this feature assists users in performing an output grouping task that would otherwise require manual effort.

6.3.1 Using the OntoUML Dataset

Data: As input data, we used the same set of models used in experiment 1. In this experiment, we executed a mining task on the synthetic dataset adopted for experiment 1 and selected 33 out of the whole set of generated patterns, where each selected pattern can be easily traced back to one of the patterns presented in Figure 7 or one of its variations (e.g., `Subkind` patterns with a missing `Subkind`). We then manually clustered all the generated patterns by assigning a cluster label to them (e.g., we labeled the `Subkind` pattern and its variations as `cluster_0`). As the final segmentation, we produced six clusters, each one representing a pattern of Figure 7.

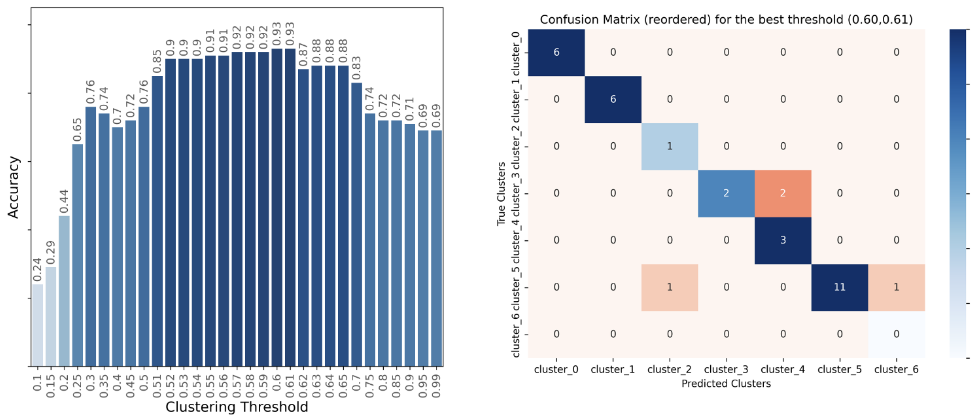


Fig. 11: Correlation *similarity threshold vs. accuracy (bar plot on the left) and true clusters vs. predicted clusters ratio for the best case when there was the number of clusters correspondence (confusion matrix on the right).*

Setup: We applied our automated clustering step testing multiple similarity threshold values, e.g., 0.1, 0.15, 0.2, 0.25. We then compared the output of each test with the manually created dataset to calculate the accuracy of the automated clustering. Note that, in multiple cases, the number of clusters generated differed from those generated by hand (e.g., when the selected threshold was 0.1, the number of clusters was 2 against 6). For this reason, we calculated the accuracy as the percentage of correctly predicted cluster assignments out of the pairs of patterns being compared. More precisely, the function we adopted calculates the accuracy by comparing pairs of elements on two lists, namely: P_i, the list of automatically clustered patterns, and P_j, the list of manually clustered patterns. For each pair of patterns (i, j), the function checks whether pattern i and j in both lists P_i and P_j are inserted the same cluster. After checking all pairs (i, j), the function returns the accuracy as the ratio of correctly predicted pairs to the total number of pairs.

When the accuracy increased significantly, we decreased the distance of the threshold values, to understand what exactly was the point of best performance (for example, instead of going from 0.5 to 0.55, we tested 0.5, 0.51, 0.52, 0.53, 0.54, ..., 0.64, etc.)

Results: We depict the outcomes in Figure 11. For the dataset we considered, the clustering component we employ in the pipeline achieves 0.93 accuracy when the adopted similarity threshold is 0.6 or 0.61. The increase in accuracy in this is due to the creation of the same number of clusters, such that the different patterns have been distinguished and the variations for each of them not deemed as different patterns. The approach incorrectly predicted only two patterns in the best case, as shown in the confusion matrix in the figure.

The confusion matrix (right) was reordered using the *Hungarian algorithm*,[16] to optimize alignment between true and predicted clusters, ensuring a clearer visualization of results and accounting for the fact that cluster labels are assigned arbitrarily by the clustering algorithm (e.g., predicted `cluster_6` could represent ground truth `cluster_1`). Rows represent true clusters (ground truth labels), columns represent predicted clusters and diagonal cells indicate correctly predicted samples, while off-diagonal cells highlight misclassification. The confusion matrix reports the results at the optimal thresholds (0.60–0.61).

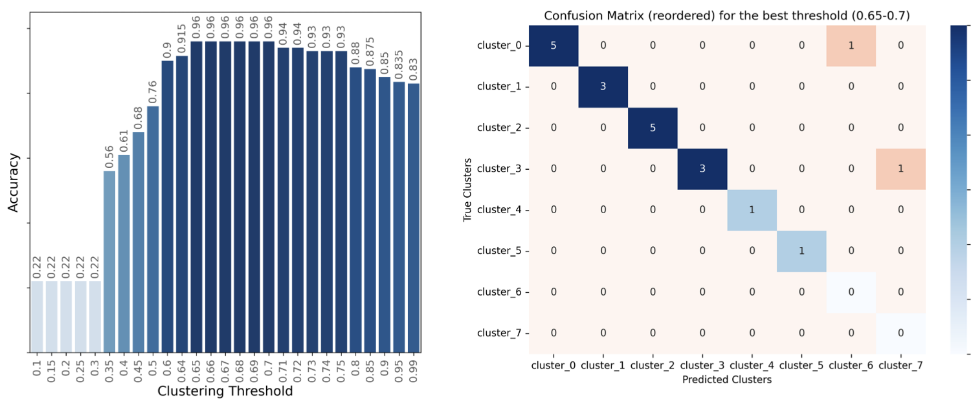
6.3.2 Using the ArchiMate Dataset

Data: For this experiment, we used the same dataset of ArchiMate models as in experiment 1 (see Section 6.1.2) and we selected 20 patterns from the mining output of the neutral trial in experiment 1. Each selected pattern corresponds to one of the patterns depicted in Figure 9 or a variation (e.g., missing an element or relationship). The chosen patterns were manually clustered into groups to serve as ground truth in our evaluation (e.g., `Chatty Service` patterns are assigned the label `cluster_0`), resulting in six total clusters, each representing a pattern of Figure 9.

Setup: We evaluated the clustering component by applying multiple similarity thresholds (e.g., 0.1, 0.2, ..., 0.9) and comparing the clustering outputs with the ground truth. Initially, to calculate clustering accuracy, we used the same pairwise comparison method as in the OntoUML experiment (see Section 6.3.1). For each pair of patterns, the predicted clustering result was compared with the ground truth to determine if the clustering algorithm accurately grouped or separated them. To refine the evaluation, we further explored thresholds within the interval where the accuracy peaked (e.g., 0.65–0.7) by testing intermediate values (e.g., 0.66, 0.67).

[16] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html

Results: The results of this experiment are shown in Figure 12. The bar plot (left) shows pair-wise accuracy (from Section 6.3.1) across various clustering thresholds. The highest accuracy, 0.96, was achieved within the range of 0.65–0.7. Beyond this range, accuracy slightly decreased due to over-clustering or merging distinct clusters. The confusion matrix (right) was reordered using the Hungarian algorithm to optimize alignment between true and predicted clusters, as in the OntoUML scenario. The results reported concern the optimal threshold between 0.65 and 0.7.



	cluster_0	cluster_1	cluster_2	cluster_3	cluster_4	cluster_5	cluster_6	cluster_7
cluster_0	5	0	0	0	0	0	1	0
cluster_1	0	3	0	0	0	0	0	0
cluster_2	0	0	5	0	0	0	0	0
cluster_3	0	0	0	3	0	0	0	1
cluster_4	0	0	0	0	1	0	0	0
cluster_5	0	0	0	0	0	1	0	0
cluster_6	0	0	0	0	0	0	0	0
cluster_7	0	0	0	0	0	0	0	0

Fig. 12: Correlation similarity threshold vs. accuracy (left) and true clusters vs. predicted clusters ratio for the best threshold (right)

6.3.3 Threats to Validity

A primary concern regarding the validity of this experiment is that we conducted the test over a limited set of patterns by considering one clustering approach only.[17] Users may vary in their way of classifying patterns, and it could be more challenging when the structures to be classified cannot be straightforwardly traced back to a reference pattern. Still, the test was enough to demonstrate how the clustering approach is reliable in distinguishing the structures that are generated and can be used to organize the data in a manner similar to what the user would adopt.

7 Application Example

To illustrate the applicability and utility of our proposed approach, we conducted a simulation of a use case. The primary objective of this simulation was to showcase

[17]Note that for the current implementation, we adopted a straightforward approach where all vectors must be of the same size, with the same dimensions. Each pattern is represented as a record in a matrix with features such as "number of nodes", "kind", "number of relations", "general", and "specific", etc. Each feature can hold different values for each pattern (for instance, if a pattern has three nodes classified as "kind", it will have "number of nodes" = 3 and "kind" = 3, while if it lacks nodes for generalizations, it will report 0 for both "specific" and "general").

the effectiveness of our approach in analyzing the practical usage of a particular modeling language: OntoUML. Our specific goal was to demonstrate how the discovery pipeline can be used to extract valuable insights that can enhance our understanding of the language, facilitate modifications, and drive improvements. Within this scope, we have used an extended version of the OntoUML dataset, comprising **143** models.[18] To ensure a substantial number of patterns for analysis, we conducted five comprehensive tests, adjusting parameters to avoid generating an overwhelming number of structures.[19] In a second step, we went through the whole set of generated patterns and we engaged in internal discussions with two experts who contributed to the design of the language.[20] The interaction with the experts occurred in two stages. First, we presented the generated patterns and asked them to identify the patterns for which it was worth having a more in-depth discussion. Second, after defining a subset of 40 patterns, we went through a discussion of each of them,[21] focusing primarily on *.i* the unexpectedness of the mined structure, and *.ii* the possibility of reusing the structure somehow. This simulation allowed us to address a new research question:

RQ4 *To what extent can the approach be used to discover new interesting structures?* This research question relates to **R1**, **R2** and **R3**, and explores the approach's capability to identify compelling structures—potentially unexpected ones—in real-world scenarios featuring a diverse set of models with varying levels of complexity. If identified, these structures can offer valuable insights for language engineers, potentially informing their future design strategies.

Next, we present four exemplary structures, taken from the subset of selected interesting patterns, which represent significant findings from our demonstration. Each example is accompanied by a brief discussion outlining the actions that could be undertaken based on our observations.

Example 1: In OntoUML, `Modes` are concepts representing particular types of properties with no structured values, which depend on their bearers [40]. According to the OntoUML specifications provided in [39] the constraints for modes are the following:

.i every `Mode` must be (directly or indirectly) connected to an association end of at least one `Characterization` relation, and

.ii the multiplicity of the characterized end (opposite to the `Mode`) must be exactly one.

Therefore, the situation in which a `Mode` is connected to two different bearers is not admitted.

[18]More information about this ever-growing dataset, with statistics about models, can be found in [15] and https://github.com/OntoUML/ontouml-models.

[19]Note that the mining process should be viewed as an iterative procedure, concluding once interesting results are obtained, so it becomes difficult to envision an optimal set of parameter *a priori*. Testing the current approach to establish optimal mining parameters and exploit the pipeline to generate a robust dataset of patterns could be a valuable objective for future research.

[20]Note that we have deliberately separated this section from the experiments section. The reason is that in this case we did not properly design an experiment, but simulated a use case with two language designers, who also participated in the requirements definition and design of the approach we are offering. That's why we talked about "demonstration" and "internal" discussion.

[21]The examples in this section can be viewed as a summary of discussions on the most relevant patterns. We added this clarification as a note at the location indicated by the reviewer.

Figure 13 shows a structure with three `Modes`, each one of them characterizing a distinct bearer, where two `Modes` are subtypes of a parent `Mode`.[22]

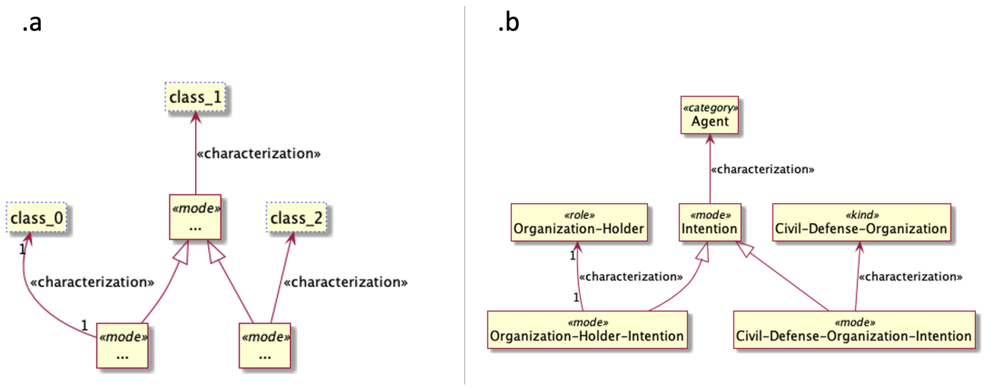


Fig. 13: Interesting structure: `Modes` *typing and multiple* `Characterizations`. *.a* found structure; *.b* example occurrence.

The information related to the occurrence of the pattern (*.b*) reveals that the reference bearers are represented by a `Role`, a `Category`, and a `Kind`, respectively. This does not yet present an actual problem; however, it may involve the generation of scenarios that are not intended. In fact, the pattern suggests checking whether `role` and `kind` are subtypes of the same concept. In fact, if this is not the case, the child `modes` might end up associated with two different `bearers` (e.g., `class_0` and `class_1` may classify different instances). This specific situation may be avoided by forcing the bearers of the child `modes` (e.g., `class_0` and `class_2`) to be subclasses of the bearer of the main parent `mode`. In conclusion, the recurring structure we found does not present an incorrect practice, but it may trigger an evolution of the language. Originally, in fact, the designers of OntoUML did not dwell on the possibility of constructing taxonomies of `modes`. However, through the results of this mining activity, emerges how the *de facto* modelers use these taxonomies (the importance of which was also discussed in [47]). This uncovered structure, then, opens the door to introducing new possible language constraints to avoid unwanted configurations.

Example 2: The structure provided by Figure 14 is significant for understanding how `Events` and their relationships are used in OntoUML.

Besides the structure we report here, many other variations having similar characteristics have been found. Consequently, a useful lesson we can learn from this is that the OntoUML module concerning events is less in control by modelers than that concerning, for example, `Kinds` or `Roles`. OntoUML has several relationships that can be used between `Events`. For example, see the common participation relationship or

[22]Nodes with dashed line denoted as `class_0` or `class_1`, i.e., without associated stereotype, are nodes that in the discovered patterns present relationships about which we have information related to the source node, but not the target node, or vice versa. This is because, as mentioned above, the graphs given as input to the mining algorithm have all relationships reified. So, for example, in Figure 13 `class_1` was created because in the pattern we had a `mode` node associated with an edge *source* to a relation *characterization*.

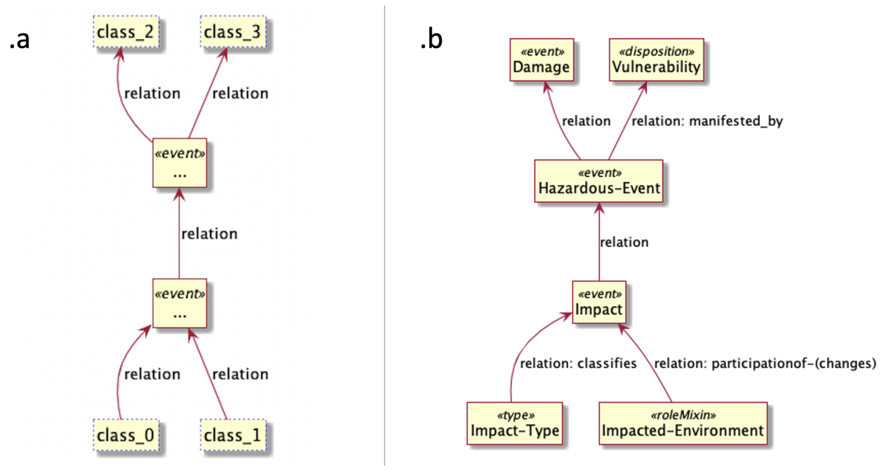


Fig. 14: Interesting structure: `Events` *and their relations. .a* found structure; *.b* example identified.

the `HistoricalDependence` relationships. The fact that the structure in Figure 14 and many similar ones do not have stereotypes for relations between `Events` can certainly be a sign that the users of the language do not properly understand these. In addition, by looking at the pattern occurrence information, one can see that some users have used relationship labels that invoke a specific stereotype (e.g., the case of `ParticipationOf` in the figure). In conclusion, this identified structure emphasizes the need for development guidelines for any related patterns for `Event` modeling.

Example 3: This example's structure still relates to `Event` modeling. Here, what we represent in Figure 15 could be an occurrence of a syntactical mistake, depending on the cardinalities of the participation relation between event and `RoleMixin`. Here, the concept of `RoleMixin` [39] is the equivalent of `Role`, but instead of being played by instances of the same `Kind` (e.g., student-person), it is played by instances of different `Kinds`. Examples of `RoleMixins` include customers and insured items, the former being played by both people and companies, whilst the latter being played by cars, houses, and paintings [48].

That being considered, the OntoUML `RoleMixin` construct requires that there are at least two identity providers involved and then at least two different `Role` instances. Thus, in what sense does a `RoleMixin` participate in an `Event`? Looking at the occurrence of the pattern, the concept of `RoleMixin` is modeled as "client" and in this sense, we may have different instances of this class that are, for instance, persons or organizations. Considering this structure, to avoid possible mistakes, the cardinalities of the relationship between the `RoleMixin` concept and the `Event` concept must be made explicit. The recurrent structure we found indicates that the modeler left this information implicit. Consequently, the possible insight that can be collected by the language engineer through the assessment of this structure is that the modeler should be guided in making explicit the connection between the identity providers and the given `Event`, possibly by providing the correct cardinalities of the associations in question.

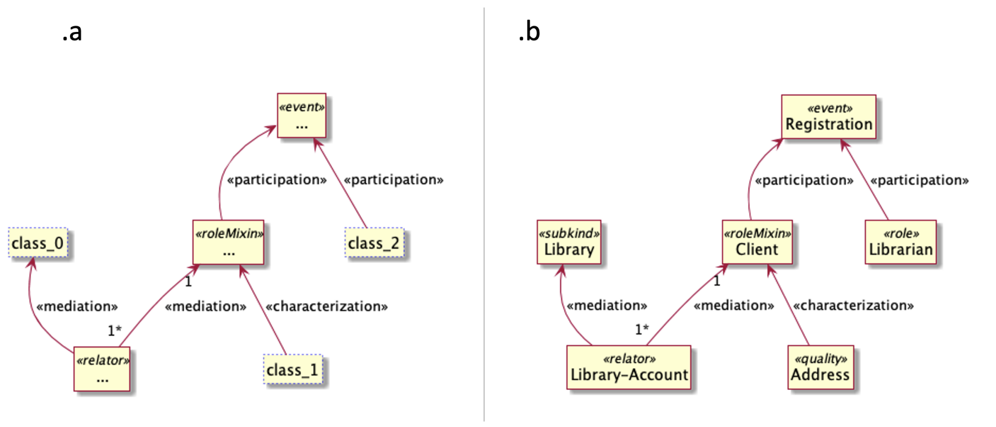


Fig. 15: Interesting structure: `Relator`, `RoleMixin` *and* `Event` *structure*. *.a* found structure; *.b* example occurrence.

8 Discussion

RQ1: Can the proposed approach generate structures encoding previously recognized interesting patterns? The answer to this question is affirmative. Experiment 1 demonstrated that already-known structures can be discovered accounting for all the constructs of which they are composed and without missing any occurrence. For the patterns we selected, we successfully discovered all classes of patterns and their occurrences. The patterns we have chosen were also useful to demonstrate how the approach can account for the core constructs in the input language. The experiment also highlighted the importance of customization options is important in reducing the number of results, thus facilitating the identification process. The importance is enlarged when the methodology is applied to extensive real-world datasets, such as the one represented by the OntoUML catalog.

RQ2: Which are the main parameters affecting the performance of the proposed approach? According to Table 5 and Table 6, the increase in pattern size and minimum support value improves the average performance but decreases the number of patterns that can be found. Unlike when the goal is to search for more frequent structures, we can conclude from this that when it is necessary to find more information or possibly unexpected information, it is necessary to find the right trade-off between effectiveness and performance. On a dataset comprising OntoUML or ArchiMate models, if we search too small and too infrequent patterns, the algorithm can produce even more than 10.000 outputs, often very similar to each other. Here, too, the role of customization steps can be essential, especially in reducing the information in the graphs to be sent as input to the mining algorithm and in eliminating redundant or unwanted outputs, thus also reducing the time of visualization and assessment.

RQ3: Is the clustering step accurate in grouping structures? Thanks to Experiment 3, we observe how our proposed clustering component provides a grouping criterion for patterns akin to manual user-performed ones. Our current configuration enables the differentiation of various structures by considering both the node count and the types of constructs involved. This capability positions it as a valuable asset during the results assessment phase. However, a point of discussion centers on the heterogeneity of the threshold values needed to achieve a satisfactory grouping, which, as we have found in other tests, can vary from case to case. We cannot assert a universally optimal threshold, such as adhering to a specific value like 5, for attaining the best grouping. Our trials have shown that optimal thresholds may range around 0.6 or 0.7 in some instances. One consistent observation, though, is that the optimal range for classification consistently falls within the mid-range of the scale. That is, thresholds between 0.1 and 0.3 or between 0.8 and 1.0 do not consistently achieve the most effective clustering. Considering the iterative nature of the discovery process, where users often experiment and fine-tune parameters to get the desired output in terms of size and type of structures, adjusting threshold values to achieve the desired clustering is also part of the iterative discovery process.

RQ4: To what extent is the approach useful for discovering new interesting structures? From the tests we did for the utility demonstration, we observed that to find unexpected modeling practices, the importance value of frequency (or minimum support) decreases. In fact, if parameters are set to find frequent structures, it is more likely that we will find already known structures, especially with a language like OntoUML, where modelers operate following predefined guidelines and patterns. This mainly has a negative effect on the overall performance of the approach. In fact, as we saw in Experiment 2, the more we decrease the frequency threshold, the longer the mining process takes. Moreover, the approach has the potential to generate patterns of a very large order of magnitude. However, this did not prevent us from collecting useful and interesting information, such as that discussed in the previous section, which can trigger a range of analyses and possibly interventions by the language engineers that they had not thought of before. Currently, our emphasis is not on exploring the interesting types and unexpected structures that can be found, as it is not the main purpose of the paper. Our understanding of classifying these outputs is growing as we consider the analyzes and operations they might trigger.

Structures triggering the definition of new patterns or anti-pattern. For instance, structures like the one represented in Figure 13 could be a good start for understanding and, possibly, devising new modeling strategies to be suggested to modelers. In the specific case of the example given, the structure of modes is a concrete case for understanding how to enable the engineers to avoid generating unintended instantiations. For example, engineers can concretely achieve this by exploiting the found structure and offering an ideal way of modeling `modes` taxonomies, or by introducing new constraints that force the creation of generalization links between the concepts characterized by the sub-`modes` and the concept characterized by the parent-`mode`.

Structures triggering a clarification of constructs and their usages. A typical example in this regard is that represented by Figure 14. Here, the pervasive absence of relation stereotypes composing structures involving events is a clear sign of how the modeling guidelines related to the area of the language devoted to these constructs (in this case, the area dependent on UFO-B [49]) can be further elaborated.

Structures that highlight possible adoptions of bad practices. In this sense, the approach is useful in unearthing new types of errors or understanding whether practices deemed to be avoided are actually adopted, and to what extent. The example provided by Figure 15 is a typical case and demonstrates the utility of the approach since it can be a strong ally in understanding how to possibly guide modelers to not omit key information or extend the language with new constraints.

9 Related Work

There is extensive literature [32, 50–52] on pattern discovery and its applications in a variety of domains, including software code [53], databases [54], educational processes [55], business processes [56], model-driven engineering (MDE) [57], EMF metamodels [58], etc. However, to the best of our knowledge, there is a significant potential for research into automatic applications that support pattern discovery in conceptual models.

In this focused area of research, the closest work to what we propose is that of Skouradaki et al. [12], who designed a pattern mining algorithm for *BPMN*. Still, the goal of our contribution is not to provide a new mining algorithm. Our emphasis predominantly lies in the combination of a well-established FSM technique with graph manipulation techniques. Furthermore, a considerable amount of effort from our side concerns the definition of an interactive process where users can participate in the discovery activities, thus affecting the reliability of the final output. Last, we designed the approach with the scope of covering different conceptual modeling languages by keeping all the functions of the approach as language-independent.

Ławrynowicz et al. [11] seek to discover domain patterns related to specific areas of information and independent of the modeling language constructs that recur across *OWL ontologies* by applying a *tree-mining* technique. They divide their contribution into two main steps, which partially resemble aspects of our strategy, namely: a transformation step, where ontology axioms are transformed into tree structures; and an association analysis step, where co-occurring axioms are extracted to discover ontology patterns. This research is applied to a set of ontologies from the *BioPortal* repository and is very similar to ours in spirit. However, our solution presents key differences. First, for the mining step, we adopted the *frequent subgraph mining* algorithm, thus involving a completely different input preparation step. Second, we devised our approach with the main goal of discovering *structural modeling patterns*, namely patterns defined simply by the combination of constructs of a modeling language. In Ławrynowicz et al.'s work, the discovered patterns concern primarily domain-specific information that may recur within or across ontologies (e.g., what are the recurrent properties of the class "person") [11]. Again, the interaction capabilities we proposed are out of their scope.

In the same direction, Lee et al. [59] seek to discover domain patterns across and within ontologies. However, to address this challenge, two different steps are adopted: a step where sub-graphs are extracted through candidate generation and chunking processes; a step where *frequent sub-graphs mining* [60] is adopted. This work also focuses on domain-specific patterns and one of its priorities is to allow the processing of large-scale knowledge graphs. Furthermore, the paper lacks instructions on how to handle an interactive discovery process.

Two other approaches that are worth mentioning are the ones presented by Ardimento et al. [61] and the one that was introduced by Fumagalli et al. [10]. Both refer directly to UML class diagrams as the reference modeling language to be mined. However, the former is more aimed at mining modeling events that occur on *Visual Paradigm*,[23] i.e., the reference editing tool, and not properly frequent subgraph structures. The second, on the other hand, is much closer to our approach and is primarily focused on using frequent item set mining, instead of frequent subgraph mining, and also does not introduce the set of facilities we have offered here to support the discovery process.

Along similar lines, particularly within the context of MDE and, more broadly, software engineering, other solutions exist that address problems similar to ours.

For instance, the work in [62] aims to enhance software maintenance by identifying design motifs, which are solutions to recurring design challenges. The authors employ two pattern-matching algorithms adapted from bioinformatics—automata simulation and bit-vector processing—to detect exact and approximate occurrences of design motifs in object-oriented code. The findings from this research underscore the potential of bioinformatics-inspired graph-matching methods to facilitate design recovery, streamline the analysis of complex software architectures, and uncover embedded design patterns.

Likewise, the study in [63] tackles the issues of recognizing design patterns in object-oriented programs. The proposed approach utilizes design motifs to identify pattern instances and filter out irrelevant matches. This strategy effectively narrows the search space, aiding the pattern discovery process.

To summarize, our approach distinguishes itself from related work through two key novel contributions. First, we empower users in their discovery process by offering a comprehensive set of interactive steps, providing guidance and assistance throughout the discovery journey. Second, we employ frequent subgraph mining, a robust established mining technique, for mining subgraphs in diagrams representing models in different languages. Our approach also accounts for all vital elements of these diagrams, including cardinalities, multiple edge labels and class labels, and multi-directionality, thus ensuring the possibility of finding patterns containing all this information.

As a final remark, the approach we propose employs various techniques across each module in the pipeline. For example, we use a technique for mining frequent structures, a technique for matching to trace domain information, a technique for transforming mining algorithm outputs into vectors, and a technique for clustering these vectors. Each of these techniques could merit its own discussion of related work, spanning fields such as "metamodel clone detection", "linguistic corpus analysis", "efficient pattern similarity computation and clustering", and "plagiarism detection".

However, we have focused our related work review on approaches that address our specific problem: the automatic support for identifying recurrent structure heuristics. Therefore, where relevant, we have included information on complementary techniques

[23] https://www.visual-paradigm.com/download/community.jsp

related to individual pipeline steps within the sections dedicated to implementation choices.

10 Final Considerations

This paper presents a practical interactive approach, whose implementation is available at https://github.com/unibz-core/CM-Mining, for automating the empirical discovery of recurrent structures in conceptual models by combining state-of-the-art graph manipulation techniques and frequent subgraph mining. Structural patterns can be exploited to identify reusable representations of bad or good modeling practices. They are typically harvested by manual labor-intensive processes, which usually take a long time to converge into pattern catalogs. Our automated process for pattern discovery facilitates the construction of pattern catalogs tailored for modeling languages. Additionally, we create a mechanism for helping language engineers to create higher-granularity primitives in that language, i.e., modeling patterns that can become part of the grammar and tools of that language [7].

Based on the encouraging results presented here, we created a list of tasks for immediate future work. First, we are going to further assess the approach to finding unexpected patterns. In order to address this, we will involve representative users and run an evaluation through usability testing. Second, we are going to test the approach with models encoded in new conceptual modeling languages, such as BPMN (**R5**).

We chose to validate the proposed approach using two state-of-the-art conceptual modeling languages, OntoUML and ArchiMate. However, our solution is designed around graph-structured data, allowing it to encode any conceptual model constructs. Furthermore, the approach is fully language-independent, apart from the initial importing and the final visualization steps, and can be adapted to various input formats.

Our focus on these two languages and their associated datasets was driven by several key factors. First, both languages include constructs—such as classes, annotations, and cardinalities—that are also prevalent in widely-used languages like UML and BPMN. Additionally, these languages are accompanied by two high-quality, scientifically validated datasets that adhere to FAIR principles. Finally, we had direct access to the language designers, who provided essential support and feedback on the functionalities and output structures required throughout our research.

References

[1] Guizzardi, G.: Theoretical Foundations and Engineering Tools for Building Ontologies as Reference Conceptual Models. Semantic Web **1**(1, 2), 3–10 (2010)

[2] Gürlebeck, K., Legatiuk, D., Nilsson, H., Smarsly, K.: Conceptual Modelling: Towards Detecting Modelling Errors in Engineering Applications. Mathematical Methods in the Applied Sciences **43**(3), 1243–1252 (2020)

[3] Harmelen, F., Teije, A.: Validation and Verification of Conceptual Models of Diagnosis. In: Validation and Verification of Conceptual Models of Diagnosis, pp.

117–128 (1997)

[4] Kayama, M., Ogata, S., Masymoto, K., Hashimoto, M., Otani, M.: A Practical Conceptual Modeling Teaching Method Based on Quantitative Error Analyses for Novices Learning to Create Error-free Simple Class Diagrams. In: 2014 IIAI 3rd International Conference on Advanced Applied Informatics, pp. 616–622 (2014). IEEE

[5] Reder, A., Egyed, A.: Model/analyzer: A Tool for Detecting, Visualizing and Fixing Design Errors in UML. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 347–348 (2010)

[6] Falbo, R.A., Guizzardi, G., Gangemi, A., Presutti, V.: Ontology Patterns: Clarifying Concepts and Terminology. In: Proceedings of the 4th International Conference on Ontology and Semantic Web Patterns - Volume 1188. WOP, pp. 14–26. CEUR-WS.org, Aachen, DEU (2013)

[7] Guizzardi, G.: Ontological Patterns, Anti-patterns and Pattern Languages for Next-Generation Conceptual Modeling. In: ER 2014, pp. 13–27 (2014)

[8] Gangemi, A., Presutti, V.: Ontology design patterns. In: Handbook on Ontologies, pp. 221–243. Springer, ??? (2009)

[9] Besheli, P.R.: The pattern of patterns: What is a pattern in conceptual modeling? In: VMBO, pp. 99–106 (2018)

[10] Fumagalli, M., Sales, T.P., Guizzardi, G.: Pattern Discovery in Conceptual Models Using Frequent Itemset Mining. In: Conceptual Modeling. ER 2022, vol. 13607, pp. 52–62 (2022). https://doi.org/10.1007/978-3-031-17995-2_4 . Springer

[11] Ławrynowicz, A., Potoniec, J., Robaczyk, M., Tudorache, T.: Discovery of Emerging Design Patterns in Ontologies Using Tree Mining. Semantic web **9**(4), 517–544 (2018)

[12] Skouradaki, M., Andrikopoulos, V., Kopp, O., Leymann, F.: RoSE: Reoccurring Structures Detection in BPMN 2.0 Process Model Collections. In: Debruyne, C., Panetto, H., Meersman, R., Dillon, T., Kühn, e., O'Sullivan, D., Ardagna, C.A. (eds.) On the Move to Meaningful Internet Systems: OTM 2016 Conferences, pp. 263–281. Springer, Cham (2016)

[13] Mitra, S., Rao, T.M.: Discovering Design Patterns in Software Behavior Models. Journal of Computing Sciences in Colleges **32**(6), 120–129 (2017)

[14] Yan, X., Han, J.: gspan: Graph-based Substructure Pattern Mining. In: 2002 IEEE International Conference on Data Mining, 2002. Proceedings., pp. 721–724 (2002). IEEE

[15] Sales, T.P., Barcelos, P.P.F., Fonseca, C.M., Souza, I.V., Romanenko, E., Bernabé, C.H., Silva Santos, L.O.B., Fumagalli, M., Kritz, J., Almeida, J.P.A., *et al.*: A FAIR Catalog of Ontology-driven Conceptual Models. Data & Knowledge Engineering **147**, 102210 (2023) https://doi.org/10.1016/j.datak.2023.102210

[16] Glaser, P., Sallinger, E., Bork, D.: EA modelset - A FAIR dataset for machine learning in enterprise modeling. In: Almeida, J.P.A., Kaczmarek-Heß, M., Koschmider, A., Proper, H.A. (eds.) The Practice of Enterprise Modeling - 16th IFIP Working Conference, PoEM 2023, Vienna, Austria, November 28 - December 1, 2023, Proceedings. Lecture Notes in Business Information Processing, vol. 497, pp. 19–36. Springer, ??? (2023). https://doi.org/10.1007/978-3-031-48583-1_2

[17] Guizzardi, G., Wagner, G., Almeida, J.P.A., Guizzardi, R.S.: Towards Ontological Foundations for Conceptual Modeling: The Unified Foundational Ontology (UFO) Story. Applied ontology **10**(3-4), 259–271 (2015)

[18] Farooq, A., Zaytsev, V.: There Is More Than One Way to Zen Your Python, 68–82 (2021) https://doi.org/10.1145/3486608.3486909

[19] Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering using UML, Patterns, and Java. Learning **5**(6), 7 (2009)

[20] Wohed, P., Aalst, W.M.P., Dumas, M., Hofstede, A.H.M., Russell, N.: Pattern-based Analysis of BPMN — An Extensive Evaluation of the Control-flow, the Data and the Resource Perspectives. Bpm center report bpm-06-17, BPMcenter.org (2006)

[21] Jiang, C., Coenen, F., Zito, M.: A Survey of Frequent Subgraph Mining Algorithms. The Knowledge Engineering Review **28**(1), 75–105 (2013)

[22] Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), vol. 1215, pp. 487–499 (1994). Santiago, Chile

[23] Faci, A., Lesot, M.-J., Laudy, C.: cgSpan: Pattern Mining in Conceptual Graphs. In: Rutkowski, L., Scherer, R., Korytkowski, M., Pedrycz, W., Tadeusiewicz, R., Zurada, J.M. (eds.) Artificial Intelligence and Soft Computing, pp. 149–158. Springer, Cham (2021)

[24] Zhang, X., Zhao, C., Wang, P., Zhou, F.: Mining Link Patterns in Linked Data. In: Gao, H., Lim, L., Wang, W., Li, C., Chen, L. (eds.) Web-Age Information Management, pp. 83–94. Springer, Berlin, Heidelberg (2012)

[25] Han, S., Ng, W.K., Yu, Y.: FSP: Frequent Substructure Pattern Mining. In: 2007 6th International Conference on Information, Communications & Signal Processing, pp. 1–5 (2007). https://doi.org/10.1109/ICICS.2007.4449818

[26] Wang, C., Wang, W., Pei, J., Zhu, Y., Shi, B.: Scalable Mining of Large Disk-Based Graph Databases. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD, pp. 316–325. ACM, New York, NY, USA (2004). https://doi.org/10.1145/1014052.1014088

[27] Xing, J., Ma, X.: DP-gSpan: A Pattern Growth-based Differentially Private Frequent Subgraph Mining Algorithm. In: 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 397–404 (2021). https://doi.org/10.1109/TrustCom53373.2021.00067

[28] Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM journal on computing **1**(2), 146–160 (1972)

[29] Silberschatz, A., Tuzhilin, A.: On Subjective Measures of Interestingness in Knowledge Discovery. In: KDD, vol. 95, pp. 275–281 (1995)

[30] García-Vico, *et al.*: A Big Data Approach for the Extraction of Fuzzy Emerging Patterns. Cognitive Computation **11**(3), 400–417 (2019)

[31] Mabroukeh, N.R., Ezeife, C.I.: A Taxonomy of Sequential Pattern Mining Algorithms. ACM Computing Surveys **43**(1), 1–41 (2010)

[32] Güvenoglu, B., Bostanoglu, B.E.: A Qualitative Survey on Frequent Subgraph Mining. Open Computer Science **8**(1), 194–209 (2018)

[33] Song, W., Truong, T., Duong, H.: Pattern Mining: Current Challenges and Opportunities. In: Proceedings of Database Systems for Advanced Applications (DASFAA) International Workshops: BDMS, BDQM, GDMA, IWBT, MAQTDS, and PMBD, vol. 13248, p. 34 (2022). Springer Nature

[34] Elkamel, A., Gzara, M., Ben-Abdallah, H.: An UML Class Recommender System for Software Design. In: 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), pp. 1–8 (2016). IEEE

[35] Ma, Z., Yuan, Z., Yan, L.: Two-level Clustering of UML Class Diagrams Based on Semantics and Structure. Information and Software Technology **130**, 106456 (2021)

[36] Guizzardi, G., Sales, T.P., Almeida, J.P.A., Poels, G.: Automated conceptual model clustering: a relator-centric approach. Software and Systems Modeling, 1–25 (2022)

[37] Rahutomo, F., Kitasuka, T., Aritsugi, M.: Semantic Cosine Similarity. In: The 7th International Student Conference on Advanced Science and Technology ICAST, vol. 4, p. 1 (2012)

[38] Verdonck, M., Gailly, F.: Insights on the Use and Application of Ontology and Conceptual Modeling Languages in Ontology-driven Conceptual Modeling. In: Conceptual Modeling. ER 2016, pp. 83–97 (2016). Springer

[39] Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. (2005). Telematica Instituut / CTIT

[40] Guizzardi, G., Fonseca, C.M., Almeida, J.a.P.A., Sales, T.P., Benevides, A.B., Porello, D.: Types and Taxonomic Structures in Conceptual Modeling: A Novel Ontological Theory and Engineering Support. Data & Knowledge Engineering **134**, 101891 (2021) https://doi.org/10.1016/j.datak.2021.101891

[41] Guizzardi, G., Benevides, A.B., Fonseca, C.M., Porello, D., Almeida, J.P.A., Sales, T.P.: UFO: Unified Foundational Ontology. Applied Ontology **17**(1), 167–210 (2022) https://doi.org/10.3233/ao-210256

[42] Robl, M., Bork, D.: Enterprise architecture management education in academia: An international comparative analysis. Complex Syst. Informatics Model. Q. **31**, 29–50 (2022) https://doi.org/10.7250/CSIMQ.2022-31.03

[43] Collins: Litmus Test. https://www.collinsdictionary.com/dictionary/english/litmus-test (Accessed in 2021-04-02)

[44] Ruy, F.B., Guizzardi, G., Falbo, R.A., Reginato, C.C., Santos, V.A.: From Reference Ontologies to Ontology Patterns and Back. Data & Knowledge Engineering **109**, 41–69 (2017) https://doi.org/10.1016/j.datak.2017.03.004

[45] Salentin, J., Hacks, S.: Towards a catalog of enterprise architecture smells. In: Gronau, N., Heine, M., Krasnova, H., Poustcchi, K. (eds.) Entwicklungen, Chancen und Herausforderungen der Digitalisierung: Proceedings der 15. Internationalen Tagung Wirtschaftsinformatik, WI 2020, Potsdam, Germany, March 9-11, 2020. Community Tracks, pp. 276–290. GITO Verlag, ??? (2020). https://doi.org/10.30844/WI_2020_Y1-SALENTIN . https://doi.org/10.30844/wi_2020_y1-salentin

[46] Smajevic, M., Hacks, S., Bork, D.: Using knowledge graphs to detect enterprise architecture smells. In: Serral, E., Stirna, J., Ralyté, J., Grabis, J. (eds.) The Practice of Enterprise Modeling - 14th IFIP WG 8.1 Working Conference, PoEM 2021, Riga, Latvia, November 24-26, 2021, Proceedings. Lecture Notes in Business Information Processing, vol. 432, pp. 48–63. Springer, ??? (2021). https://doi.org/10.1007/978-3-030-91279-6_4

[47] Guizzardi, G., Fonseca, C.M., Benevides, A.B., Almeida, J.P.A., Porello, D., Sales, T.P.: Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In: Conceptual Modeling. ER 2018, vol. 11157, pp. 136–150 (2018). https://doi.org/10.1007/978-3-030-00847-5_12 . Springer

[48] Sales, T.P., Guizzardi, G.: Ontological Anti-patterns: Empirically Uncovered Error-prone Structures in Ontology-driven Conceptual Models. Data & Knowledge Engineering **99**, 72–104 (2015)

[49] Guizzardi, G., Wagner, G., Almeida Falbo, R., Guizzardi, R.S., Almeida, J.P.A.: Towards Ontological Foundations for the Conceptual Modeling of Events. In: Conceptual Modeling. ER 2013, pp. 327–341 (2013). Springer

[50] Fournier-Viger, P., Lin, J.C.-W., Kiran, R.U., Koh, Y.S., Thomas, R.: A Survey of Sequential Pattern Mining. Data Science and Pattern Recognition **1**(1), 54–77 (2017)

[51] Gan, W., Lin, J.C.-W., Fournier-Viger, P., Chao, H.-C., Tseng, V.S., Philip, S.Y.: A Survey of Utility-oriented Pattern Mining. IEEE Transactions on Knowledge and Data Engineering **33**(4), 1306–1327 (2019)

[52] Fournier-Viger, P., Gan, W., Wu, Y., Nouioua, M., Song, W., Truong, T., Duong, H.: Pattern Mining: Current Challenges and Opportunities. In: Rage, U.K., Goyal, V., Reddy, P.K. (eds.) International Workshops on Database Systems for Advanced Applications (DASFAA), pp. 34–49. Springer, Cham (2022)

[53] Pham, H.S., Nijssen, S., Mens, K., Nucci, D.D., Molderez, T., Roover, C.D., Fabry, J., Zaytsev, V.: Mining Patterns in Source Code using Tree Mining Algorithms, 471–480 (2019) https://doi.org/10.1007/978-3-030-33778-0_35

[54] Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu, C.-W., Tseng, V.S.: SPMF: A Java Open-Source Pattern Mining Library. The Journal of Machine Learning Research **15**(1), 3389–3393 (2014)

[55] Bogarín, A., Cerezo, R., Romero, C.: A Survey on Educational Process Mining. WIREs Data Mining and Knowledge Discovery **8**(1), 1230 (2018) https://doi.org/10.1002/widm.1230

[56] Tiwari, A., Turner, C.J., Majeed, B.: A Review of Business Process Mining: State-of-the-art and Future Trends. Business Process Management Journal **14**(1), 5–22 (2008)

[57] Pescador, A., Garmendia, A., Guerra, E., Cuadrado, J.S., Lara, J.: Pattern-based development of domain-specific modelling languages. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 166–175 (2015). IEEE

[58] Babur, Ö., Constantinou, E., Serebrenik, A.: Language usage analysis for emf metamodels on github. Empirical Software Engineering **29**(1), 23 (2024)

[59] Lee, K., Jung, H., Hong, J.S., Kim, W.: Learning Knowledge Using Frequent Subgraph Mining from Ontology Graph Data. Applied Sciences **11**(3), 932 (2021)

[60] Ramraj, T., Prabhakar, R.: Frequent subgraph mining algorithms — a survey. Procedia Computer Science **47**, 197–204 (2015)

[61] Ardimento, P., Aversano, L., Bernardi, M.L., Carella, V.A., Cimitile, M., Scalera, M.: UML Miner: A Tool for Mining UML Diagrams. In: Proceedings of the 26th International Conference on Model-Driven Engineering Languages and Systems (2023). https://doi.org/10.1109/MODELS-C59198.2023.00014

[62] Kaczor, O., Guéhéneuc, Y.-G., Hamel, S.: Identification of design motifs with pattern matching algorithms. Information and Software Technology **52**(2), 152–168 (2010)

[63] Guéhéneuc, Y.-G., Guyomarc'h, J.-Y., Sahraoui, H.: Improving design-pattern identification: a new approach and an exploratory study. Software Quality Journal **18**, 145–174 (2010)