Mining Java Memory Errors using Subjective Interesting Subgroups with Hierarchical Targets

Youcef Remil, Anes Bendimerad, Mathieu Chambard, Romain Mathonat, Marc Plantevit, Mehdi Kaytoue

Mining Java Memory Errors using Subjective Interesting Subgroups with Hierarchical Targets

Youcef Remil*†, Anes Bendimerad†, Mathieu Chambard‡, Romain Mathonat†, Marc Plantevit§ and Mehdi Kaytoue*†
*Univ Lyon, INSA Lyon, CNRS, LIRIS UMR 5205, F-69621, Lyon, France
†Infologic, 99 avenue de Lyon, F-26500 Bourg-Lès-Valence, France
‡Ecole Nationale Supérieure de Rennes, FR-35170, Bruz, France
§EPITA Research Laboratory (LRE), FR-94276, Le Kremlin-Bicetre, France

***Abstract*—Software applications, particularly Enterprise Resource Planning (ERP) systems, are crucial to the day-to-day operations of many industries, where it is essential to maintain these systems effectively and reliably. In response, Artificial Intelligence for Operations Systems (AIOps) has emerged as a dynamic framework, harnessing cutting-edge analytical technologies like machine learning and big data to enhance incident management procedures by detecting, predicting, and resolving issues while pinpointing their root causes. In this paper, we leverage a promising data-driven strategy, dubbed Subgroup Discovery (SD), a data mining method that can automatically mine incident datasets and extract discriminant patterns to identify the root causes. However, current SD solutions have limitations in handling complex target concepts with multiple attributes organized hierarchically. We illustrate this scenario by examining the case of Java out-of-memory incidents among several possible applications. We have a dataset that describes these incidents, including their topology and the types of Java objects occupying memory when it reaches saturation, with these types arranged hierarchically. This scenario inspires us to propose a novel Subgroup Discovery approach that can handle complex target concepts with hierarchies. To achieve this, we design a new Subgroup Discovery framework along with a pattern syntax and a quality measure that ensure the identified subgroups are relevant, non-redundant, and resilient to noise. To achieve the desired quality measure, we use the Subjective Interestingness model that incorporates prior knowledge about the data and promotes patterns that are both informative and surprising relative to that knowledge. We apply this framework to investigate out-of-memory errors and demonstrate its usefulness in root-cause diagnosis. Notably, this paper stands out as a contribution to both Data Mining and Java memory analysis research. To validate the effectiveness of our approach and the quality of the retrieved patterns, we present an empirical study conducted on a real-world scenario from our ERP system.**

***Index Terms*—AIOps, Java Memory Analysis, Data Mining, Subgroup Discovery, Subjective Interestingness.**

I. Introduction

Industries are rapidly moving toward digitization thanks to the notable progress made in AI and software development. At the heart of this revolution, Enterprise Resource Planning (ERP) software systems are directly connected to factories and their equipments. They provide a large panel of tools and data to users across different services of a company and significantly help them to achieve their daily tasks. With the increasing importance of these software systems in numerous industry activities, there is a growing need to ensure their optimal performance. Collecting data from various sources within the ERP ecosystem allows organizations to gain valuable insights into operational efficiency and identify areas for improvement. In this context, Artificial Intelligence for IT Operations (AIOps) emerges as a game-changing technology that refers to the application of advanced data-driven algorithms including machine learning and big data analytics to intelligently improve, strengthen, and automate a wide range of IT operations [1]–[4]. Therefore, organizations are turning to AIOps in order to detect, prevent, diagnose, and rapidly mitigate high-impact incidents [5]. One of the most common incidents encountered is related to `OutOfMemory` errors, which occur when the memory allocated to the software is fully utilized, often due to a memory leak caused by some specific bug. Engineers usually rely on tools that provide statistics about the memory content at the time of the error, which aids in identifying the root cause. For example, the `jmap` command depicts the memory consumed by each class in a Java Virtual Machine, as shown in Fig. 2. Moreover, these classes are organized *hierarchically* in packages. For example, the class `LinkedHashMap` in Line 9 is part of the package `java.util`, which is itself included in the package `java`. By analyzing the histogram, analysts can identify classes that consume significantly more memory than expected, which may be the underlying root-cause of memory saturation.

Analyzing histograms can be an overwhelming task for several reasons. First, the analyst may lack a clear understanding of what constitutes "normal" size consumption for each class, requiring significant experience or reference histograms with a "healthy" memory consumption. In Fig. 2, for example, the `[C` class may not be the cause of the incident despite being the most consuming class, as this value may be its usual size. Secondly, some memory incidents are not related to a single class but to a software feature that impacts several classes belonging to specific packages. This raises the question of how to concisely identify the suspicious packages and/or classes without redundancy, especially when dealing with many hierarchy levels. Finally, analysts often have to inspect a vast dataset of histograms related to various incidents described by their contexts (e.g., software version and type) as well as

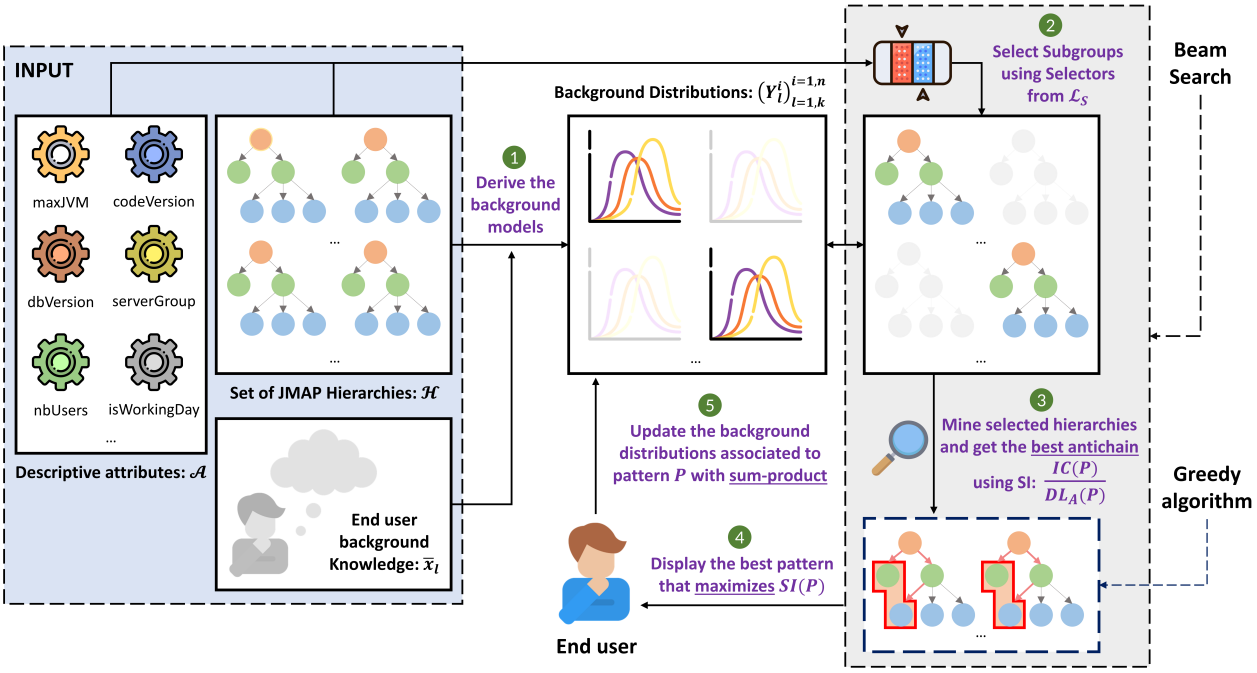


Figure 1: Overview of our Subjectively Interesting Subgroup Discovery Framework with hierarchical target concepts.

`jmap` histograms describing memory content.

In this case, the analyst seeks to find contexts that significantly co-occur with specific kinds of memory errors to pinpoint the root cause for several incidents at once. Existing methods [6]–[9] primarily focus on diagnosing Java memory problems separately, without analyzing large sets of incidents simultaneously to discover common patterns. Also, most of them address particularly the problem of memory leak detection, whereas there are other possible factors that can cause `OutOfMemory` incidents. Our goal is to address these challenges, by providing a generic approach that extracts useful information from such datasets.

Mining datasets to identify subgroups that exhibit some property of interest is known as Subgroup Discovery (SD) [10], [11]. Among many possibilities, SD aims to find patterns describing subsets of data where the distribution of a target variable deviates from the "norm". However, none of existing SD approaches have considered the case when the target concept consists in multiple attributes organized as hierarchies, such as `jmap` histograms. To address this limitation, we introduce a novel and generic SD setting that elegantly handles this case. We employ Subjective Interestingness framework [12] to model the information dependency between hierarchical attributes, and assess informativeness of patterns. This framework has been successfully instantiated in several data mining tasks [13]–[15]. Its strength lies in its ability to include the user's prior knowledge about the data to find unexpected patterns. It also allows to iteratively update the interestingness model to account for information already transmitted to the user during the mining task. We characterize subgroups with specific subsets of target attributes called *antichains* (set of hierarchically incomparable elements). This antichain constraint allows us to avoid redundancy within the same pattern, as hierarchically related attributes often transmit the same information. Different from the method proposed in [16] which extracts contrastive attributes but from a single hierarchy, our solution mines a dataset of many hierarchies described by additional contextual attributes. Our solution is then applied to analyze a dataset of `jmap` histograms and contextualize memory errors, although it remains generic and suitable for numerous SD tasks with hierarchical target attributes.

Figure 1 provides an overview of the entire process underlying our proposed approach. The input dataset comprises incident data, including `jmap` histograms and other relevant attributes that contextualize each incident. Additionally, the approach leverages prior information pertaining to the dataset under analysis. As an example, these priors could consist of `jmap` histograms computed from "healthy" servers, serving as benchmarks for expected class sizes. Our approach utilizes these priors to establish background distributions that represent "healthy" memory usage patterns. Subsequently, it sifts through the incident dataset to identify the most informative subgroups, pinpointing suspicious classes and packages that exhibit significant deviations from the established background distribution. Once this subgroup is identified, it is communicated to the analyst. Following this, the background model is updated to assimilate this newfound information, which the analyst is already aware of. This iterative process can be repeated as necessary, allowing for a refined understanding of

```
> jmap -histo
 num   #instances      #bytes  class name
----------------------------------------------
   1:      121908    11267384  [C
   2:       19054     2895288  [I
   3:      118284     2838816  [Ljava.lang.Object
   4:       26595     2474152  [Ljava.lang.Class
   5:       20056     2236120  [B
   6:        9627     1618344  [J
   7:        1027     1599864  java.util.HashMap$Node
   8:       46906     1138200  [Ljava.util.HashMap$Node
   9:       12942      766720  java.util.LinkedHashMap
  10:       19168      528346  org.hibernate.metamodel.internal.BasicTypeImpl
```

Figure 2: Example of a `jmap` histogram.

the dataset's anomalies and patterns. Each of the framework phases will be discussed in detail in the upcoming sections.

Contributions. Our contribution is three-fold: (1) Unlike existing Java memory analysis methods that address individual memory issues within specific use cases, we present a comprehensive approach that tackles a wide range of related incidents stemming from various root causes, which allows for a more holistic diagnosis. (2) We present a pioneering approach to address a novel Subgroup Discovery challenge where the target concept consists of multiple attributes organized in a hierarchy by introducing an adapted pattern syntax and a new interestingness measure rooted in Subjective Interestingness, which properly guides the search for informative and non-redundant patterns, along with a simple yet effective beam-search algorithm to search for interesting subgroups. (3) Employing both qualitative and quantitative analysis, we showcase the effectiveness of our approach by utilizing a real-world ERP dataset to mine contextualized `jmap` histograms, thereby diagnosing the root causes of memory errors.

II. Methodology

We performed our analysis on a dataset comprising approximately 4,000 Java memory snapshots, also known as `Java memory heap dumps`, which were collected from over 350 servers over a three-month period. Each memory snapshot represents the instantiated objects residing in the Java virtual machine heap of a specific server at a specific moment. Each snapshot is associated with a hierarchical structure that defines a directed acyclic dependency graph, grouping classes and/or sub-packages within the same package, along with information about the size of their instantiated objects. Furthermore, we enriched these snapshots with additional descriptive features, including environmental variables, resulting in a wealth of properties that provide contextual information for Subgroup Discovery.

A. Raw Data

Java Memory Heap Dumps. The analysis of Java memory plays a crucial role in monitoring the performance of Java applications. Java virtual machine (`JVM`) uses the heap memory to store objects created by an application,

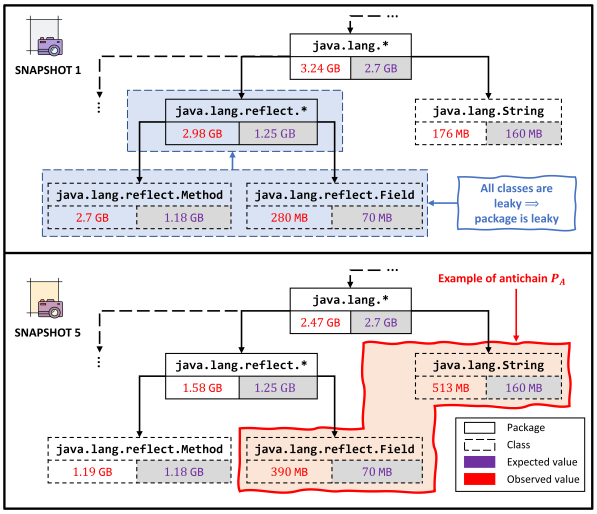


Figure 3: Example of `JMAP` sub-hierarchy retrieved from a specific memory snapshot with some class/package sizes.

and garbage collection is performed to reclaim memory resources that are no longer used. However, the `JVM` may not be able to free up memory if objects still retain references, which can lead to memory saturation and cause the `java.lang.OutOfMemoryError` exception to be thrown. This error can be due to insufficient allocated heap size, a coding error that causes memory leaks, or loading a large number of objects into memory simultaneously. To obtain specific memory-related statistics, we leverage the command-line utility `jmap` with the `-histo` option. This allows us to obtain a class-wise histogram that displays the number of instantiated objects in the heap per class, the total memory used for each of these objects, and the fully qualified class name. We use this histogram for each memory snapshot to build a hierarchy that groups the classes and/or sub-packages recursively into their parent packages. The sub-hierarchy illustrated in Fig. 3 demonstrates that the package size is the total size of all its sub-packages or classes. For instance, the `java.lang.` package contains the sub-package `java.lang.reflect.` as well as the class `java.lang.String`.

Topology information for contextualization. Each memory snapshot is accompanied by additional descriptive characteristics. Particularly, these features describe the execution environment and the software component in which the exception `OutOfMemoryError` is triggered. For instance, our ERP software is characterized by its application identifier, its software version, its declination among main server families such as sales, factory, etc. The JVM can be parameterized by the flag `Xmx` which specifies the maximum memory allocation pool for the `JVM`, the `Xms` to indicate the initial memory allocation pool. Information on when the memory collapsed is also provided (e.g., whether it is a working day).

Table I: Toy Example of a dataset $(\mathcal{O}, \mathcal{A}, \mathcal{H})$. Gray cells indicates that the size values are larger than what was expected.

$\mathcal{O}$	**Descriptive attributes $\mathcal{A}$**				**Size of instantiated objects w.r.t. packages in MB** represented with hierarchies $\mathcal{H}$ (see example for snapshots o_1 and o_5 in Fig. 3)				
	softType	softVersion	Xmx	weekDay	J.L.⋆	J.L.reflect.⋆	J.L.reflect.Field	J.L.reflect.Method	J.L.String
o_1	Sales	V_3	$4.2e+09$	True	3242	2980	280	2700	176
o_2	Sales	V_3	$2.3e+09$	False	3296	3003	322	2678	355
o_3	EDI	V_1	$6.4e+09$	True	2305	1474	264	1210	163
o_4	Factory	V_1	$1.8e+09$	False	2217	1481	386	1095	480
o_5	Factory	V_2	$2.4e+09$	True	2475	1582	390	1192	513
o_6	Manager	V_2	$5.3e+09$	True	2016	1258	56	1202	140
o_7	Sales	V_3	$2.4e+09$	True	3398	2814	320	2494	402
o_8	Factory	V_3	$8.2e+09$	False	2715	1326	84	1200	147
o_9	Sales	V_3	$6.4e+09$	True	2430	1577	412	1165	120
o_{10}	Sales	V_1	$4.5e+09$	True	2570	1283	68	1215	422

B. A Unified Data Model

We create a dataset $\mathcal{D} = (\mathcal{O}, \mathcal{A}, \mathcal{H})$ that unifies the different data sources mentioned above. $\mathcal{O} = \{o_i\}_{1 \leqslant i \leqslant n}$ is a set of objects that correspond to memory snapshots indexed by the pairs `(server;timestamp)`. $\mathcal{A} = (a_j)_{1 \leqslant j \leqslant m}$ is a vector of descriptive attributes used to contextualize the snapshots, and $\mathcal{H} = \{H_i\}_{1 \leqslant i \leqslant n}$ is a set of hierarchies constructed from `jmap` histograms. These hierarchies group classes and/or sub-packages that belong to the same package and include the size of their instantiated Java objects.

Each attribute a can be represented as a mapping $a : \mathcal{O} \longrightarrow R_a$, where R_a is called the domain of attribute a. R_a is given by $\mathbb{R}$ if a is numerical, by a finite set of categories C_i if a is categorical, or by $\{0, 1\}$ if a is Boolean. Table I provides a dataset with 10 objects $\mathcal{O} = \{o_1, ..., o_{10}\}$ corresponding to 10 memory snapshots generated during `outOfMemoryError` exceptions. Each object is described by 4 attributes and referenced by a hierarchy $H \in \{H_1, ..., H_{10}\}$. For example, the attribute `softType` is nominal and has 4 possible values. The attribute `Xmx` is numerical, and the attribute `weekDay` is a Boolean that indicates whether the memory crash occurred on a working day. Fig. 3 illustrates with a typical example of sub-hierarchical elements that have varying values of both H_1 and H_5, which highlight the integer-valued attributes associated with class sizes and their corresponding packages. For instance, the package `java.lang.reflect` in H_1 has a size of 2.98 GB and contains only two classes, namely, `java.lang.reflect.Method` (size of 2.7 GB) and `java.lang.reflect.Field` (size of 280 MB).

C. Hierarchical Target Concepts

We consider the scenario where the concepts of interest are defined as a set of positive integer-based attributes that are structured hierarchically. These hierarchically organized concepts are generally referred to as *counters* based on their observed values. These counters represent either the size of the classes (which are located at the leaves of the hierarchy) or packages (which are located at internal nodes). The root node represents the size of the heap at the time of the memory crash. We formally define a hierarchy $H \in \mathcal{H}$ as follows.

Definition 1 (Hierarchy)**.** A hierarchy $H_i \in \mathcal{H}$ is defined as a tuple $H_i = (E^{(i)}, \leqslant, \langle e_1, x_1^{(i)} \rangle)$ for $i \in [\![1, n]\!]$ where:

- $E^{(i)} = \{\langle e_1, x_1^{(i)} \rangle, ..., \langle e_k, x_k^{(i)} \rangle\}$ is a set of k items (nodes or concepts) with their counters. For convenience, We use sometimes $E = \{e_1, ..., e_k\}$ to refer to the set of items without their counters.
- $\leqslant$ is a partial order relation defined over this set E, indicating the relationship of predecessors between hierarchically linked concepts,
- $\forall e \in E : e_1 \leqslant e$ (the item e_1 is called the root of $\mathcal{H}$)
- there is only one path from the root e_1 to any other item:

$$\forall e_j, e_k, e_l \in E : e_j \leqslant e_l \wedge e_k \leqslant e_l \Longrightarrow e_j \leqslant e_k \vee e_k \leqslant e_j.$$

For example, in Fig. 3, both of the following relations hold: `(java.lang.⋆)` $\leqslant$ `(java.lang.reflect)` and `(java.lang.⋆)` $\leqslant$ `(java.lang.reflect.Field)`.

Moreover, we assume that the counter value x_l of a concept e_l is always larger or equal than the value of its successors. We introduce the following operations used throughout the remainder of this paper. Given $S \subseteq E$, we have:

- Precedessors operator $\Uparrow$, and successors operator $\Downarrow$ as:

$$\Uparrow S = \{e \in E \mid \exists e' \in S : e \leqslant e'\},$$
$$\Downarrow S = \{e \in E \mid \exists e' \in S : e' \leqslant e\},$$

- Strict predecessor relation: $e_j \prec e_k \Leftrightarrow e_j \leqslant e_k \wedge e_j \neq e_k$,
- The direct successor relation $\prec$ as: $e_j \prec e_k \iff \Downarrow \{e_j\} \cap \Uparrow \{e_k\} = \{e_j, e_k\}$. Also, if $e_j \prec e_k$, we use the notation $\pi_k = j$ to refer to the index of the only direct parent of e_k ($e_j = e_{\pi_k}$),

The counter value of a concept e_l for an object $o_i \in \mathcal{O}$ is denoted using the discrete random variable $X_l^{(i)} \in \mathbb{N}$. If a particular value of a concept e_l is empirically observed for the object $o_i \in \mathcal{O}$, it is denoted in the hierarchy H_i by $\hat{x}_l^{(i)}$.

D. Contrastive Antichains as patterns

We first introduce the concept of contrastive antichains as interesting patterns based on a single hierarchy. These patterns are comprised of a subset of hierarchically disjoint concepts that are informative and non-redundant. In our case study, we aim to concisely inform developers about suspicious classes and packages. To evaluate the interestingness of a pattern, we rely on prior knowledge about counters, such as developers' rough estimates of the space occupied by classes in the heap. For example, in Fig. 3, developers expect the size of the `java.lang.string` class to be around 160 MB based on

the analysis conducted on healthy servers. Thus, discovering that this class takes up 513 MB in snapshot o_5 is surprising.

Providing the user with such concepts can be interesting. However, because one counter's information affects the expectation of other hierarchically related concepts, one intuition is to recursively aggregate the interesting concepts at the same level into a higher level of the hierarchy. This approach is shown in Fig.3, where, for example, instead of listing both the leaking classes `java.lang.reflect.Method` and `java.lang.reflect.Field`, we only provide their parent package `java.lang.reflect`. Another intuition is to only include non-comparable concepts in the pattern (i.e., no concept is a predecessor or successor of any other concept in the same pattern). We refer to this set of concepts by an *antichain*. Thus, an antichain denoted as P_A ensures that the information provided is not redundant. For each identified concept $e_l \in P_A$ for a specific object $o_i \in \mathcal{O}$, a contrastive antichain pattern informs the user about the value $\hat{x}_l^{(i)}$. Yet, users generally tend to memorize only the order of magnitude indication instead of precise values. Hence, we refer to the counters in patterns with the scale $\lfloor log_2(\hat{x}_l^{(i)}) \rfloor$.

Definition 2 (Contrastive Antichains)**.** Given a hierarchy $H_i = (E^{(i)}, \leqslant, \langle e_1, x_1^{(i)} \rangle)$ with pairs of concepts and their values $\langle e_l, x_l^{(i)} \rangle \in E^{(i)}$, a contrastive antichain pattern $P_A \subseteq E$ is a subset of concepts that form an antichain w.r.t. $\leqslant$, i.e., $\forall e_j, e_k \in P_A$: $e_j \leqslant e_k \implies e_j = e_k$, with the integers $\lfloor \log(\hat{x}_l^{(i)}) \rfloor$ describing the scale of the values of their counters.

E. Need to Characterize Subgroups with a Common Antichain

Developers often analyze a large set of objects (memory snapshots) at once. Providing contrastive antichains for each individual object can be overwhelming, and many objects may share hierarchical concepts that have similar properties. We need to simultaneously characterize a subset of objects that are together associated to a contrastive antichain pinpointing suspicious classes or packages. An interesting example in Table I is the subgroup $\{o_2, o_4, o_5, o_7\}$ containing memory snapshots from virtual machines with a `Xmx` flag value not exceeding $2.5e + 09$. All these snapshots exhibit unexpectedly high memory consumption of classes forming the following antichain: {`java.lang.reflect.Field`, `java.lang.String`}. Identifying interesting subgroups along with their antichains necessitates a unified *pattern language*. Since there is an extremely large number of subgroups that can be derived from combinations of descriptive attributes and target antichains, an automatic Subgroup Discovery approach can prove invaluable by exploring a set of candidate hypotheses and using an quality function to score subgroups and identify the best of them. We exploit the subjective interestingness framework (SI) proposed in [12]. This framework makes it possible to iteratively incorporate the new information provided to the user when communicating a subgroup to her, to avoid communicating subgroups with redundant information. For instance, suppose the user is presented with the subgroup $\{o_1, o_2, o_7, o_9\}$ defined by Sales servers of Version 3 associated with an antichain consisting of only the package `java.lang.reflect`. Subsequently, the user is presented with another subgroup containing objects $\{o_2, o_4, o_5, o_7\}$, associated with the antichain {`java.lang.reflect.Field`, `java.lang.String`}. Although the second pattern is interesting, it becomes less surprising when the user is aware of the first pattern, because $\{o_2, o_7\}$ are already associated with the concept `java.lang.reflect`, which is a package that already includes `java.lang.reflect.field`. Hence, our approach should ignore this pattern and suggest a more restrictive one, such as the subgroup that covers only $\{o_4, o_5\}$ with the description ($\texttt{Xmx} < 2.5e+09 \wedge \text{softType} = \text{Factory}$).

III. Subjectively Interesting Subgroups with Contrastive Antichains in Hierarchies

To efficiently exploit Subgroup Discovery with hierarchical target concepts, we need to address many challenges: (1) handling the complex data structure including both subgroup descriptions and antichains, (2) assessing pattern interestingness related to a specific subgroup and its retrieved antichain, (3) defining a mining algorithm that is scalable and can identify subgroups and antichains that maximize the proposed measure of interestingness, (4) implementing an effective mechanism to incrementally update the user's background knowledge.

A. Pattern Language

We consider a pattern language defined as a pair $\mathcal{L} = (\mathcal{L}_S, \mathcal{L}_A)$ such that $\mathcal{L}_S$ is the *subgroup pattern language* defined over descriptive attributes $\mathcal{A}$, and $\mathcal{L}_A$ is the *antichain pattern language* defined over the concepts E from $\mathcal{H}$. A pattern $P \in \mathcal{L}$ is then given as $P = (P_s, P_A)$, where $P_s \in \mathcal{L}_S$ is a constrained selector of a subset of objects using their descriptive attribute values and $P_A \in \mathcal{L}_A$ is a retrieved antichain from E. In more details, the subgroup pattern language is defined as $\mathcal{L}_S = \times_{j=1}^{|\mathcal{A}|} Sel_j$, where Sel_j is a selector applied over an attribute $a_j \in \mathcal{A}$ that describes a set of objects based on their attribute values (e.g., it is given by the set of all possible intervals in $\mathbb{R}$ if a_j is numerical). Hence, $P_s \in \mathcal{L}_S$ is then given by a set of restrictions over each descriptive attribute i.e., $P_s = (Sel_j)_{1 \leqslant j \leqslant m}$. These patterns are ordered from the most general to the most restrictive by an order relation $\sqsubseteq$. More precisely, for two patterns $P_s = (Sel_j)_{1 \leqslant j \leqslant m} \in \mathcal{L}_S$ and $P'_s = (Sel'_j)_{1 \leqslant j \leqslant m} \in \mathcal{L}_S$, we have: $P_s \sqsubseteq P'_s \Leftrightarrow \forall j \in [\![1, m]\!] \, (Sel_j \supseteq Sel'_j)$. On the other hand, the antichain pattern language is defined as the set of all possible antichains that can be derived from E i.e., $\mathcal{L}_A = \{P_A \subseteq E \mid \forall e_l, e_k \in P_A \text{: } e_l \leqslant e_k \implies e_l = e_k\}$.

Subgroup patterns and objects. A subgroup pattern $P_s = (Sel_j)_{1 \leqslant j \leqslant m}$ is referred to as *covering* an object $o \in \mathcal{O}$ iff $\forall j \in [\![1, m]\!] : a_j(o) \in Sel_j$. A subgroup pattern P_s *covers* a set $O \subseteq \mathcal{O}$ iff it covers each object $o \in \mathcal{O}$. Using the *cover* concept, we define the function $\delta(O) \in \mathcal{L}_S$ which gives the most restrictive subgroup pattern that covers a set of objects O: $\forall P_s \in \mathcal{L}_S$, P_s covers O iff $P_s \sqsubseteq \delta(O)$. For a given subgroup

pattern P_s, the set of all objects covered by P_s is referenced by the *extent* concept: $ext(P_s) = \{o \in \mathcal{O} \mid P_s \sqsubseteq \delta(o)\}$ [17].

Definition 3 (Subgroup)**.** A subgroup is any subset of objects $s \in \mathcal{O}$ that can be selected using a pattern P_s over descriptive attributes $\mathcal{A}$. The set of all possible subgroups is denoted $\mathcal{S} = ext(\mathcal{L}_S) = \{ext(P_s) \mid P_s \in \mathcal{L}_S\}$. In other terms, a subgroup is a set of objects that can be characterized with some restrictions of attributes, turning it interpretable to the user.

B. Subjective Interestingness Measure

We design a subjective interestingness measure to assess the quality of each pattern $P = (P_s, P_A)$. This function measures its surprisingness when contrasted with some background distribution that represents user priors about the data. Therefore, we need to formally model the prior beliefs about each counter $\langle e_l, x_l^{(i)} \rangle \in E^{(i)}$. We will present these counters through the probability distributions $Pr(X_l^{(i)} = x_l^{(i)})$.

Background Distributions. For each concept in the hierarchy, we assume that we have a reference that derives either an approximation of its value or its proportion to other concepts that are hierarchically dependent on it. In our case, we compute `jmap` histograms on normally behaving servers to derive the expected values of the size or size proportion of each class and/or package in the memory heap. Our goal is to represent the expected value x_l of each concept $e_l \in E$, as well as the expectations conditioned on the parent except for the root, i.e., $x_l \mid x_{\pi_l}$. As an example, the `java.lang.String` class is expected to have an average size of approximately 160MB, and the package `java.lang.reflect` is expected to contain about 95% of the `java.lang.reflect.Method` class.

Given these two constraints, there are an infinite number of possible solutions for defining probability distributions for all concepts. To address this, we follow the approach of [12] and consider only the distributions that maximize entropy, meaning that they do not introduce additional assumptions beyond the explicitly specified expectations that would reduce the entropy. As noted in [16], this approach results in geometric probability distributions for each random variable $X_l^{(i)}$, hence:

Property 1. The marginal distribution for each random variable $X_l^{(i)}$ is geometric, and it is given as:

$$\Pr(X_l^{(i)} = x_l) = \left(1 - \frac{1}{1 + \bar{x}_l}\right)^{x_l} \cdot \frac{1}{1 + \bar{x}_l}.$$

Interestingness of a Pattern. Now that a probability distribution has been defined to model the user's background knowledge, we need to evaluate the antichain P_A on each object of the subgroup $o \in ext(P_s)$ with respect to its probability distribution. This is necessary to efficiently determine the most interesting and surprising patterns that contradict the background beliefs or the previously discovered findings (i.e., after iteratively updating its primary knowledge). To assess the score of a given pattern $P = (P_s, P_A) \in \mathcal{L}$, we propose a new quality measure rooted in the framework of subjective interestingness SI [12]. $SI(P)$ is defined as the ratio between the information content of the pattern P and its description length: $SI(P) = \frac{IC(P)}{DL(P)}$.

The *Information Content* (IC) measures the amount of information communicated to the user. It is defined as the negative log probability under the background distribution: $IC(P) = -\log(\Pr(P))$. However, calculating the information content of such complex pattern can be challenging, especially when dealing with multiple probability distributions of the same concept associated with different objects in the subgroup. To address this, we aim to quantify the information gain provided by an effective aggregation of counters $\langle e_l, \hat{x}_l^{(i)} \rangle$ in P_A, by evaluating the objects of the subgroup under their respective probability distributions (which might have been updated in previous iterations). A straightforward solution would be to use the mean value of each concept in the subgroup. However, the mean is too sensitive to outliers and does not provide a complete overview of the subgroup values w.r.t. a concept, as no assumptions are made about the subgroup variance.

To overcome this issue, we came out with the idea of calculating the cumulative distribution function from the quantile q_α of order α that indicates the value below which a certain percentage of the subgroup values fall, for each concept. The hyperparameter $(0 < \alpha < 1)$ is selected based on the quality of the subgroups returned. For instance, if the quantile of order 0.25 is considered, then our solution informs the user that 75% of the subgroup values are greater than this quantile. To better understand this approach, let's consider the subgroup $s = \{o_1, o_2, o_7, o_9\}$ with corresponding values $\{2980, 3003, 2814, 1577\}$ for the package `java.lang.reflect`. Assuming that all objects have the same geometric probability distribution with a mean of 1250MB (in the first iteration), we can use the cumulative distribution function under the quantile of order 0.25 to calculate the probability $\Pr(X^{(i)}_{\texttt{java.lang.reflect.}} \geqslant 2814) = 0.08$, which is interesting. In other words, for the subgroup defined as (softType = Sales $\wedge$ softVersion = V_3), 75% of its objects have a size that is greater than or equal to 2814MB for the `java.lang.reflect.` package. This information is valuable and surprisings since it deviates from what we would expect based on its probability distribution.

As previously stated in Sec. II-D, the information content is communicated to the user by transmitting the scales of the values, instead of the exact values. As a consequence, we define the new random variables $Y_l^{(i)} = \lfloor \log_2(\hat{X}_l^{(i)}) \rfloor$. Concretely, the IC of a pattern $P \in \mathcal{L}$ is given as follows:

$$\begin{aligned} IC(P) = IC\left((P_s, P_A)\right) &= -\log\left(\prod_{o_i \in s}\prod_{e_l \in P_A} \Pr(Y_l^{(i)} \geqslant {q_{\alpha}}_l^{s})\right), \\ &= -\sum_{o_i \in s}\sum_{e_l \in P_A} \log\left(\Pr(Y_l^{(i)} \geqslant {q_{\alpha}}_l^{s})\right), \\ &= -\sum_{o_i \in s}\sum_{e_l \in P_A} \log\left((1-p_l)^{2^{{q_{\alpha}}_l^{s}}} - (1-p_l)^{2^{{q_{\alpha}}_l^{s}+1}}\right). \end{aligned}$$

where ${q_{\alpha}}_l^{s}$ is the quantile of order α of the values $\{y_l^{(i)}\}_{o_i \in s}$ for the concept $e_l \in P_A$

The *description length* (DL) is a measure of the complexity involved in communicating a pattern P to a user. In our case we propose to compute it based on both the subgroup pattern P_s and the antichain P_A. When communicating the antichain, items closer to the root e_1 are more likely to be familiar to the user and easier to interpret. For instance, it is simpler to communicate the package `java.lang` (2^{nd} level) than the class `java.lang.reflect.Method` (4^{th} level). On the other hand, a subgroup with only a few selectors is easier to interpret as it helps to quickly pinpoint the root cause. In order to characterize as many objects as possible in a subgroup that is distinguished by an interesting antichain, we avoid linear penalization of the subgroup size. Hence, the DL is given as:

$$DL(P) = DL_s(P_s) \cdot DL_A(P_A)$$
$$= (\beta \cdot \log(|s|) + \gamma \cdot \|P_s\|) \cdot \left(\eta \cdot (\sum_{e_l \in P_A} 1 + \log(|\Uparrow \{e_l\}|)) \right)$$

with β, γ and η are hyperparameters to weight each part of the DL according to the user preferences.

C. Updating the Background Knowledge

When conveying a pattern to the user, her background knowledge model must be updated to take into consideration the new piece of information. The communicated pattern values are likely to become the new expected values, and therefore, the probability distributions must also be updated accordingly. In the following, let $\mathcal{R} \subseteq \mathcal{L}$ be the set of patterns that have already been observed up to the i$^{\text{th}}$ iteration, that is, $\mathcal{R} = (P_s^1, P_A^1), ..., (P_s^{(i)}, P_A^{(i)})$. We refer to the quality of the pattern P assuming the user has knowledge of $\mathcal{R}$ as:

$SI(P \mid \mathcal{R}) = \frac{IC(P|\mathcal{R})}{DL(P)} = \frac{-\log(\Pr(P|\mathcal{R}))}{DL(P)}$

The probability $\Pr(P \mid \mathcal{R})$ represents the likelihood of pattern P appearing in the data given that the user is aware of the quantile of order α for the subgroup values of each concept for all previously communicated patterns $P' = (P'_s, P'_A) \in \mathcal{R}$. In other words, instead of knowing the exact value of each object, the user only knows that its probability being higher than ${q_\alpha}_l^{s'}$ is $(1-\alpha)$. Thus, we update the probability distribution $\Pr(Y_l^{(i)} = y_l)$ as follows:

$$\Pr(Y_l^{(i)} = y_l) = \begin{cases} \Pr(Y_l^{(i)} = y_l) \cdot \frac{1-\alpha}{\Pr(Y_l^{(i)} \geqslant {q_\alpha}_l^{s'})}, & \text{if } Y_l^{(i)} \geqslant {q_\alpha}_l^{s'} \cdot \\ \Pr(Y_l^{(i)} = y_l) \cdot \frac{\alpha}{1-\Pr(Y_l^{(i)} \geqslant {q_\alpha}_l^{s'})}, & \text{otherwise.} \end{cases}$$

For example, assuming that the user has been given the subgroup $s' = \{o_1, o_2, o_7, o_9\}$, whose antichain contains the package `java.lang.reflect` and that the first quartile (i.e., $\alpha = 0.25$) has been used to retrieve useful patterns, then the new probability distribution for each of the objects in s' must verify: $\Pr(Y^{(i)}_{java.lang.reflect} \geqslant \lfloor \log_2(2814) \rfloor) = 0.75$.

The hierarchical organization of the concepts implies that updating the probability distribution of a particular concept will have a direct and recursive impact on its predecessors and successors. For instance, if the user becomes certain (with a probability of 75%) that the size of the `java.lang.reflect.` package is larger than 2814, then it follows that the size of its parent, `java.lang.`, must also be larger than 2814 with a probability greater than 75%, since $X_{\pi_l} > X_l$. These dependencies between the random variables can be represented using a Bayesian tree, which is a graphical model. To propagate the impact of updating some random variables to all nodes in the hierarchy, we use the sum-product inference algorithm [18].

D. Mining Interesting Patterns

The process of finding the most interesting patterns in our context is very costly, since the computational complexity of a subgroup discovery task is known to be prohibitive and results from the huge size of the search space $|\mathcal{S}|$ that increases exponentially. Besides, each generated subgroup pattern $P_s \in \mathcal{L}_f$ has to be evaluated with the set of all possible antichains L_A. Moreover, the interestingness measure used in this paper SI is not monotonic (i.e., it is not straightforward to derive non-trivial bounds on its values to prune some uninteresting patterns), which implies that exhaustive search is not a feasible strategy to be adopted. We employ optimization procedures that are commonly used in both scenarios (i.e., enumeration of subgroup patterns and the search for contrastive antichains). We derive `SCA-Miner`, a heuristic approach that uses beam search to generate at each level of the lattice the most interesting subgroups with their associated antichain which is retrieved with a greedy search algorithm w.r.t. the subjective interestingness measure.

The proposed algorithm is outlined in Algorithm 1 (`SCA-MINER`), which is an iterative approach, that aims at each iteration to provide the user with an interesting pattern P. The algorithm starts by updating the model using the sum-product method to incorporate the user's previously acquired knowledge $\mathcal{R}$. Then, it employs a beam search strategy to enumerate the subgroup patterns and use a greedy search algorithm to derive the associated antichain that maximizes the SI. The algorithm continues until the beam search yields an empty pattern or the maximum size of the best patterns collection $\mathcal{P}$ is reached. The beam search systematically explores the conjunctions of selectors by expanding a limited set of patterns that have the largest SI so far. It evaluates the subgroup patterns on their set of hierarchies to extract the best associated contrastive antichain. In this phase, a greedy search method is used to build an antichain for a specific subgroup pattern. This process is repeated on each level in the beam search, where only the most promising patterns are maintained. The mining process stops when all possible selectors are explored or a chosen stopping criterion is met (e.g., the search depth). The algorithm outputs the best pattern found throughout the search.

IV. Experiments

In this section, we present the experimental study we conducted to evaluate the quality of results provided by `SCA-Miner` * to analyze Java memory errors reported by our ERP software. We aim to assess whether the approach is capable of identifying interesting patterns based on Subjective

*https://github.com/RemilYoucef/sca-miner

Algorithm 1: SCA-Miner

Input: the dataset: $\mathcal{D}_{crash} = (\mathcal{O}, \mathcal{A}, \mathcal{H})$, $width$: the number of most promising subgroups per level, $depth$: the maximum depth to explore in the lattice, $threshold$: a threshold on the number of patterns.

Output: $\mathcal{P}$: An ordered collection of patterns $P \in \mathcal{L}$ sorted based on iteratively updated SI.

$\mathcal{P} \leftarrow \langle\rangle$
$\mathcal{R} \leftarrow \varnothing$
repeat
  // Update the model with the sum-product algorithm
  // and derive the probabilities $\Pr(Y_l^{(i)} = \hat{y}_l^{(i)} \mid \mathcal{R})$:
  Sum-product($\mathcal{H}$, $\mathcal{R}$)
  // Get the best subgroup along with its associated contrastive antichain:
  $(P_s, P_A) \leftarrow$ BeamSearch($\mathcal{D}$, $width$, $depth$, $\mathcal{R}$)
  if $P_s \neq \varnothing$ *and* $P_A \neq \varnothing$ **then**
    $\mathcal{P}.append((P_s, P_A))$
    $\mathcal{R} \leftarrow \mathcal{R} \cup (P_s, P_A)$
until $P_s = \varnothing$ *or* $P_A = \varnothing$ *or* $|\mathcal{P}| = threshold$;

Interestingness and whether the update of the background model is effective. Additionally, we assess the interpretability and relevance of the identified patterns in pinpointing potential root causes regarding our case study.

A. Experimental Setup and Methodology

Datasets and hyperparameters. Our experimental study involved analyzing more than 4,000 Java memory heap dumps collected over a 3-month period from approximately 350 servers. To establish reference values for the average heap space usage of each class or package, we generated a separate dataset of heap dumps from healthy servers at the beginning of each week. We only considered the top 200 classes in each histogram associated with a heap dump, as these are often the ones that retain the most heap space. The resulting dataset comprised 3,320 memory snapshots, each described by 14 descriptive attributes, and mapped into hierarchies to contextualize each subgroup and antichain. We used the readily available data mining tool Pysubgroup [19] to extend the beam search algorithm to fit our pattern language and measure of interest. We set the beam width to 50 and the number of selective selectors to 4, and displayed the top 20 patterns based on Subjective Interestingness (SI). We set the quantile of order 20 to inform users about 80% of the subgroup data values for a specific concept. The hyperparameters associated with the DL function were set empirically as $\beta = 0.8, \gamma = 0.2$, and $\eta = 1$.

Baselines. While there are no approaches in the literature that specifically support hierarchical Subgroup Discovery with interesting target concepts, we consider some baselines in our study to highlight the benefits of SCA-MINER's novel features, including the hierarchical structure and the new interestingness measure, as well as its capability to iteratively update the user background knowledge to avoid redundancy. First, we compare with the SI approach, which returns the best results according to our interestingness measure, but does not iteratively update the background model. Next, we compare with Customized WRAcc (CWRAcc), which adapts the widely-used WRAcc measure [20] to our problem, measuring the deviation of the subgroup mean value from the mean value of the entire dataset regarding a target concept. Specifically, $CWRAcc(P_s, P_A, \theta) = \frac{1}{|P_A|^\theta} \sum_{e_l \in P_A} (\frac{1}{|s|} \sum_{o_i \in s} \hat{x}_l^{(i)} - \bar{x}_l)$. We run the Beam Search algorithm under the same hyperparameters as SCA-MINER. We also compare against the KL divergence, as performed in [21], to measure the difference between the observed probability distributions for the contextualized memory snapshots and the expected probability distribution, at each level of the hierarchy. Finally, we perform a post-processing step PP for each of these baselines to eliminate redundant patterns according to the Jaccard coefficient, as performed in the work of [22].

Comparative Metrics: To evaluate the effectiveness of SCA-MINER compared to baselines, we utilize two metrics: **Average Contrast** and **Redundancy**. The first metric measures the contrast between the observed subgroup values $\hat{x}_l^{(i)}$ and their expected values $\bar{x}_l$ for each retrieved pattern. We propose using the contrast metric defined as follows for a pattern P: $contrast(P) = \sum e_l \in P_A \frac{\frac{1}{|s|}(\sum o_i \in s \hat{x}^{(i)} l) - \bar{x} l}{\frac{1}{|s|} \sum o_i \in s \hat{x}^{(i)} l}$. The second metric, redundancy, measures the redundancy between patterns communicated to the user while accounting for the redundancy between hierarchically related concepts of different antichains with respect to the same subgroup elements. To accomplish this, we propose the following redundancy measure evaluated on a set of patterns: $redund(\mathcal{P}) = \frac{\sum P \in \mathcal{P} max P' \in \mathcal{P} \setminus P Jaccard(s,s') \cdot \frac{|e_l \in P_A | \exists e_k \in P'_A : e_l \leqslant e_k \vee e_k \leqslant e_l|}{|P_A \cup P'_A|}}{|\mathcal{P}|}$.

B. Results interpretation

Comparative evaluation. Fig. 4 displays the comparative measures used to evaluate SCA-MINER against the baseline techniques discussed earlier. The SI measure produces larger contrast values in general (with or without update) compared to the CWRAcc and KL-Divergence measures. This is due to the fact that CWRAcc typically retrieves antichains containing nodes with higher counters, which do not necessarily result in contrastive patterns, as their average contrast is remarkably low when compared to that of the SI measure. Nonetheless, the contrast values of the non-updated SI tend to be higher than those of the updated SI because the algorithm often focuses on the same top patterns, generating a high level of redundancy. This redundancy is demonstrated in the bar graphs, where more than 60% of the patterns are redundant, which makes the process less surprising for the end user. Similarly, when using the CWRAcc and KL-Divergence measures, the resulting patterns often contain redundancy. Even after applying post-processing to the final pattern set, there is still a much higher level of redundancy when compared to using the updated SI measure (only about 4% redundancy).

Illustrative results. We present the top 4 patterns discovered by SCA-MINER from our dataset in Fig.5. Each pattern is accompanied by a description of its corresponding subgroup and the number of memory snapshots it covers. The red color

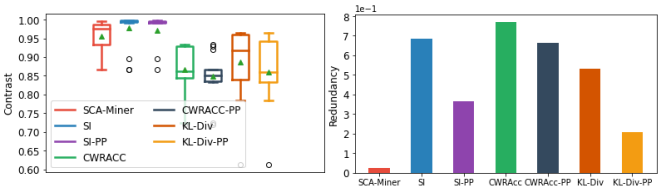


Figure 4: Comparison of contrast and redundancy between top patterns of SCA−MINER against the baselines.

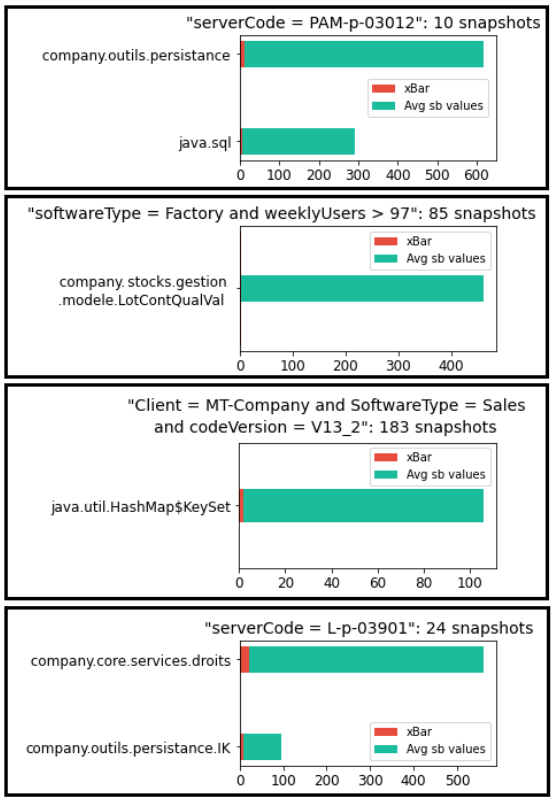


Figure 5: Top patterns returned by SCA−MINER

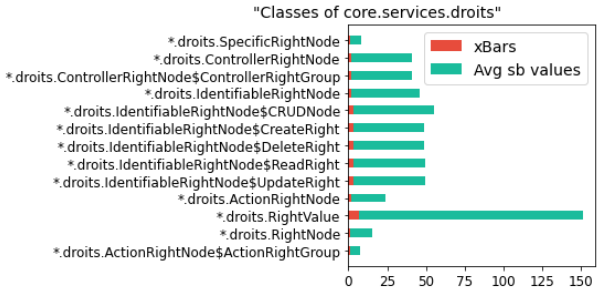
Table II: Statistics related to antichains that belong to the top patterns retrieved with SCA−MINER

Top k	Antichains	$\bar{x}$	Min	$q_{0.2}$	Avg	Max
Top 1	company.outil.persistance	13	568	579	606	659
	java.sql	7	267	271	284	305
Top 2	company.stock.gestion. modele.LotContQualVal	3	1	344	458	749
Top3	java.util.HashMap$KeySet	2	1	104	104	106
Top 4	company.core.services.droits	21	104	482	541	730
	company.outils.persistance.IK	8	80	82	88	118

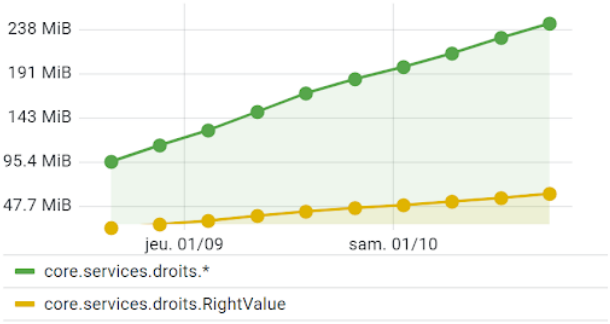
in the figure represents the expected value of the concept in the antichain, while the green color indicates the average observed values of the subgroup with respect to this concept. We chose to report the mean value in the charts since it is more interpretable, intuitive, and comparable to the expected value. However, in TableII, we also provide all the statistics related to the top 4 patterns, including the minimum and maximum value, and $0.2-$order quantile, which further reveals the properties of the subgroup distributions. Overall, our approach has successfully identified interesting and surprising over-expressed patterns for all the extracted patterns. These patterns are diverse, non-redundant, and cover large subgroups (e.g., 85 and 183 memory snapshots in the second and third pattern). Additionally, our approach is capable of outputting generic packages as well as single classes when relevant.

The first pattern discovered by the algorithm SCA−MINER highlights that the server PAM−p−03012 experienced a consistent increase in heap space usage for two packages: company.outils.persistance and java.sql. The latter package includes several classes such as sql.Timestamp and sql.BigDecimal, indicating that the memory saturation is due to improper usage of the Direct SQL API in the source code. This API allows direct access to the database from the source code, bypassing the Hibernate (Object-Relational-Mapping) layer. Although it avoids loading Java objects mapped to the database in memory, it can cause memory saturation with SQL objects if not properly handled. The second component of the antichain is the package company.outils.persistance, which covers classes used to identify objects with primary keys, further strengthening the hypothesis of excessive object loading with the Direct SQL API. This hypothesis was confirmed by reviewing past maintenance resolution tickets, some of which occurred shortly after memory saturation. It's important to note that although this incident highlights a memory crash problem, it is not a memory leak, as it occurs rapidly due to an unhandled use case. This confirms that our method is not limited to specific issues and can diagnose any memory-related problem.

The antichain's second pattern is noteworthy because it highlights the class LotContQualVal, which is highly contrastive for all Factory servers, with more than 97 weekly users. This pattern reveals that the frequent use of this particular class in Factory servers tends to cause memory saturation when the number of users exceeds a certain threshold, which is considerably high in the context of an ERP for industries. Despite having the best Information Content (IC), this pattern is ranked second because it has a lower generality and a relatively high depth compared to other patterns (i.e., $DL_A$is larger). On the other hand, Despite having a smaller depth, pattern 3 has been ranked lower than pattern 2 because it has more selectors (3), making it less interpretable than pattern 2. The Factory servers are crucial because they manage the core of the factory and our clients' production. The LotContQualVal class is instrumental in identifying functionality with an unusual problem, particularly those related to product quality features. This problem leads to a quick saturation event, and many



(a) Size of the classes belonging to `services.droits` package in pattern 4.



(b) Heap consumption of `services.droits` during 50 days in a leaky server.

Figure 6: Explanations of one of SCA−MINER patterns

maintenance tickets raised by our clients have identified this class as one of the crucial elements in root-cause analysis.

Pattern 4 highlights a prevalent issue in our ERP, which is memory leaks caused by the `RightValue` class. Therefore, in this pattern, we provide our experts with a more general solution that identifies memory leak problems across all classes in the `services.droits` package, rather than just a specific class. This is shown in Fig.6a, where we compare the expected value with the subgroup average value for all relevant classes. Through further analysis with our experts, we discovered a growing memory leak that had gone unnoticed for several weeks (Fig.6b), which was not detected by other supervision tools. The antichain package helped pinpoint the source of the bug in the source code so that it could be fixed as a temporary solution. However, to prevent this problem from recurring, we established a systematic memory snapshot control to monitor the trend of the package size.

V. Related work

Java Memory Analysis. When it comes to Java memory analysis, identifying classes that leak memory and cause an `OutOfMemoryError` with a large heap size can be an overwhelming task for developers. One common approach involves analyzing heap dumps with available tools such as `Eclipse-MAT` [23] to determine which objects are consuming abnormally large amounts of heap space. However, detecting and diagnosing memory leaks has been the subject of a wide range of related work, including both static and dynamic approaches. Static methods involve formulating leak detection as a reachability problem by identifying value flows from the source `malloc` to the sink `free` [24] or detecting static liveness regions [25]. Dynamic methods, on the other hand, rely on either growing types [8] (i.e., types with a growing number of run-time instances) or object staleness [7], [26] (i.e., the elapsed time since the last use of an object). However, these methods and/or tools require either the entire heap dumps or the complete code flow diagram as input, which can be prohibitively expensive in industrial production scenarios. Furthermore, unlike our approach, these solutions are applied separately to analyze specific memory problems, rather than considering a generic approach that handles simultaneously a large set of memory incidents.

Subgroup Discovery. Subgroup Discovery is recognized as an important technique to derive insights or make discoveries from data. This well-established data mining task aims to identify interpretable local subgroups that foster some property of interest. A popular case consists in binary target concepts [20] that seeks to find subsets of data such that the proportion of a specific target class is significantly higher than expected. A considerable amount of literature on Subgroup Discovery approaches consider complex target attributes such as categorical [27], numerical [28], [29] and multi-target [30], [31]. Furthermore, various approaches have been proposed to generalize Subgroup Discovery to complex data structures, e.g., sequential data [32] and graphs [33]. Hierarchies had been early taken into account in the pattern syntax [34]. However, they remain under-exploited on the target side. None of the existing techniques has exploited this framework for hierarchical target concepts. Our proposed method is based on the `FORSIED` framework [35], which formalizes the data exploration process as an interactive exchange of information between the model and data analyst, incorporating the analyst's prior belief state. The framework employs Subjective Interestingness (SI) [12], which has been successfully used to assess subgroups in various structures such as n-ary relations [36] and graphs [14], [15]. In [16], the authors investigated the discovery of subjectively contrastive attributes within a single hierarchy. Our work extends this approach by mining subgroups across multiple hierarchies.

VI. Conclusion and Perspectives

As today's IT environments continue to evolve, growing larger and more intricate, they generate a diverse range of heterogeneous data that necessitates efficient analysis for incident management. This demands effective data-driven approaches rooted in the concept of AIOps, capable of extracting insightful patterns from such data. In this paper, we presented `SCA-MINER`, the first data mining algorithm enabling the discovery of interesting and contextualized subgroups from data with targets anchored in a hierarchy along with descriptive attributes, while taking into account non-redundancy and interestingness using the SI framework. Thanks to a real-world dataset of Java memory incidents provided by our ERP supervision team, we showcase several actionable patterns,

some of which have been seamlessly integrated into our rule-based maintenance engine. Moving forward, we intend to consistently employ the algorithm for the discovery of new and interesting rules, bugs, and memory leaks. Additionally, we plan to extend this methodology to other suitable use cases, such as the analysis of microservices traces that document hierarchy-like call dependencies among microservices. In parallel, we aim to enhance the computational efficiency of the mining algorithm by incorporating and comparing alternative heuristic approaches (e.g., MCTS, genetic algorithms, etc.). For the algorithm's practical usability in daily operations, akin to regular SQL workload analyses conducted in [37], particular emphasis should also be placed on the temporal dimension when maintaining prior knowledge.

REFERENCES

[1] Y. Dang, Q. Lin, and P. Huang, "Aiops: real-world challenges and research innovations," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 4–5.

[2] J. Bogatinovski, S. Nedelkoski, A. Acker, F. Schmidt, T. Wittkopp, S. Becker, J. Cardoso, and O. Kao, "Artificial intelligence for it operations (aiops) workshop white paper," *arXiv preprint arXiv:2101.06054*, 2021.

[3] S. Becker, F. Schmidt, A. Gulenko, A. Acker, and O. Kao, "Towards aiops in edge computing environments," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 3470–3475.

[4] P. Notaro, J. Cardoso, and M. Gerndt, "A survey of aiops methods for failure management," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 12, no. 6, pp. 1–45, 2021.

[5] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu *et al.*, "Towards intelligent incident management: why we need it and how we make it," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1487–1497.

[6] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proceedings of the 10th ESEC-FSE*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005.

[7] C. Jung, S. Lee, E. Raman, and S. Pande, "Automated memory leak detection for production use," in *36th ICSE*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014.

[8] M. Weninger, E. Gander, and H. Mössenböck, "Analyzing data structure growth over time to facilitate memory leak detection," in *Proceedings of the 2019 ACM/SPEC ICPE*, V. Apte, A. D. Marco, M. Litoiu, and J. Merseguer, Eds. ACM, 2019.

[9] V. Sor and S. N. Srirama, "Memory leak detection in java: Taxonomy and classification of approaches," *J. Syst. Softw.*, vol. 96, pp. 139–151, 2014.

[10] W. Klösgen, "Explora: A multipattern and multistrategy discovery assistant," in *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996, pp. 249–271.

[11] M. Atzmueller, "Subgroup discovery," *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, vol. 5, no. 1, pp. 35–49, 2015.

[12] T. De Bie, "Maximum entropy models and subjective interestingness," *Data Mining and Knowledge Discovery*, vol. 23, no. 3, pp. 407–446, 2011.

[13] S. Kapoor, D. K. Saxena, and M. van Leeuwen, "Online summarization of dynamic graphs using subjective interestingness for sequential data," *Data Min. Knowl. Discov.*, vol. 35, no. 1, pp. 88–126, 2021.

[14] J. Deng, B. Kang, J. Lijffijt, and T. D. Bie, "Mining explainable local and global subgraph patterns with surprising densities," *Data Min. Knowl. Discov.*, vol. 35, no. 1, pp. 321–371, 2021.

[15] A. Bendimerad, A. Mel, J. Lijffijt, M. Plantevit, C. Robardet, and T. D. Bie, "Sias-miner: mining subjectively interesting attributed subgraphs," *Data Min. Knowl. Discov.*, vol. 34, no. 2, pp. 355–393, 2020.

[16] A. Bendimerad, J. Lijffijt, M. Plantevit, C. Robardet, and T. D. Bie, "Contrastive antichains in hierarchies," in *Proceedings of the 25th ACM SIGKDD*, 2019, pp. 294–304.

[17] B. Ganter and S. O. Kuznetsov, "Pattern structures and their projections," in *ICCS*, ser. Lecture Notes in Computer Science, vol. 2120. Springer, 2001, pp. 129–142.

[18] C. M. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. Springer, 2007.

[19] F. Lemmerich and M. Becker, "pysubgroup: Easy-to-use subgroup discovery in python," in *ECML/PKDD*, vol. 11053. Springer, 2018, pp. 658–662.

[20] N. Lavrac, B. Kavsek, P. A. Flach, and L. Todorovski, "Subgroup discovery with CN2-SD," *JMLR*, vol. 5, pp. 153–188, 2004.

[21] M. van Leeuwen and A. J. Knobbe, "Diverse subgroup set discovery," *Data Min. Knowl. Discov.*, vol. 25, no. 2, pp. 208–242, 2012.

[22] M. Atzmüller and F. Puppe, "Semi-automatic refinement and assessment of subgroup patterns," in *Proceedings of the 21st AAAI*. AAAI Press, 2008, pp. 323–328.

[23] eclipse, "Eclipse memory analyzer (mat)." [Online]. Available: https://www.eclipse.org/mat/

[24] B. Hackett and R. Rugina, "Region-based shape analysis with tracked locations," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on POPL*, J. Palsberg and M. Abadi, Eds. ACM, 2005, pp. 310–323.

[25] R. Shaham, E. K. Kolodner, and S. Sagiv, "Automatic removal of array memory leaks in java," in *Compiler Construction, 9th International Conference, CC ETAPS*, ser. Lecture Notes in Computer Science, D. A. Watt, Ed., vol. 1781. Springer, 2000, pp. 50–66.

[26] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 17:1–17:28, 2013.

[27] H. Grosskreutz, S. Rüping, and S. Wrobel, "Tight optimistic estimates for fast subgroup discovery," in *ECML/PKDD 2008*, vol. 5211. Springer, 2008, pp. 440–456.

[28] F. Lemmerich, M. Atzmueller, and F. Puppe, "Fast exhaustive subgroup discovery with numerical target concepts," *Data Min. Knowl. Discov.*, vol. 30, no. 3, pp. 711–762, 2016.

[29] M. Boley, B. R. Goldsmith, L. M. Ghiringhelli, and J. Vreeken, "Identifying consistent statements about numerical data with dispersion-corrected subgroup discovery," *Data Min. Knowl. Discov.*, vol. 31, no. 5, pp. 1391–1418, 2017.

[30] J. Lijffijt, B. Kang, W. Duivesteijn, K. Puolamäki, E. Oikarinen, and T. D. Bie, "Subjectively interesting subgroup discovery on real-valued targets," in *34th ICDE*. IEEE Computer Society, 2018, pp. 1352–1355.

[31] W. Duivesteijn, A. Feelders, and A. J. Knobbe, "Exceptional model mining - supervised descriptive local pattern mining with complex target concepts," *Data Min. Knowl. Discov.*, vol. 30, no. 1, pp. 47–98, 2016.

[32] H. Grosskreutz, B. Lang, and D. Trabold, "A relevance criterion for sequential patterns," in *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013*, 2013, pp. 369–384.

[33] M. Kaytoue, M. Plantevit, A. Zimmermann, A. A. Bendimerad, and C. Robardet, "Exceptional contextual subgraph mining," *Mach. Learn.*, vol. 106, no. 8, pp. 1171–1211, 2017.

[34] M. Kamber, J. Han, and J. Chiang, "Metarule-guided mining of multi-dimensional association rules using data cubes," in *Proceedings of the 3rd International Conference on KDD*, 1997, pp. 207–210.

[35] T. D. Bie, "Subjective interestingness in exploratory data mining," in *Advances in Intelligent Data Analysis XII - 12th IDA*, 2013, pp. 19–31.

[36] J. Lijffijt, E. Spyropoulou, B. Kang, and T. D. Bie, "P-n-rminer: A generic framework for mining interesting structured relational patterns," in *2015 IEEE International Conference on DSAA*, 2015, pp. 1–10.

[37] Y. Remil, A. Bendimerad, R. Mathonat, P. Chaleat, and M. Kaytoue, ""what makes my queries slow?": Subgroup discovery for SQL workload analysis," in *36th IEEE/ACM International Conference on ASE*. IEEE, 2021, pp. 642–652.