Investigating Order Information in API-Usage Patterns: A Benchmark and Empirical Study

Ervina Çergani[1], Sebastian Proksch[2], Sarah Nadi[3] and Mira Mezini[1]
[1]*Software Technology Group, Technische Universität Darmstadt, Darmstadt, Germany*
[2]*Software Evolution and Architecture Lab, University of Zürich, Zürich, Switzerland*
[3]*Department of Computing Science, University of Alberta, Alberta, Canada*

Keywords: API Usage Pattern Types, Code Repositories, Events Mining, Empirical Evaluation, Benchmark.

Abstract: Many approaches have been proposed for learning Application Programming Interface (API) usage patterns from code repositories. Depending on the underlying technique, the mined patterns may (1) be strictly sequential, (2) consider partial order between method calls, or (3) not consider order information. Understanding the trade-offs between these pattern types with respect to real code is important in many applications (e.g. code recommendation or misuse detection). In this work, we present a benchmark consisting of an *episode mining* algorithm that can be configured to learn all three types of patterns mentioned above. Running our benchmark on an existing dataset of 360 C# code repositories, we empirically study the resulting API usage patterns per pattern type. Our results show practical evidence that not only do partial-order patterns represent a generalized super set of sequential-order patterns, partial-order mining also finds additional patterns missed by sequence mining, which are used by a larger number of developers across code repositories. Additionally, our study empirically quantifies the importance of the order information encoded in sequential and partial-order patterns for representing correct co-occurrences of code elements in real code. Furthermore, our benchmark can be used by other researchers to explore additional properties of API patterns.

1 INTRODUCTION

Application Programming Interfaces (APIs) provide effective means for code reuse. Client developers of an API must be aware on how to correctly use it in order to avoid errors. An *API usage pattern* encodes a set of API methods that are frequently used together, optionally complemented by constraints like the order in which methods must be called. API patterns are used as the basis for various applications such as API documentation generation (Montandon et al., 2013), automated code completion (Nguyen et al., 2012), bug or anomaly detection (Wasylkowski et al., 2007), and code search (Zhong et al., 2009a).

Many techniques have been proposed to learn three kinds of patterns from code repositories (Robillard et al., 2013): (1) *No-order patterns* are unordered sets of frequently used methods (e.g.,(Negara et al., 2014; Nguyen et al., 2016)) and encode that calls of methods, say a, b, and c, frequently co-occur in code, but do not include information about the order of calls. (2) *Sequential-order patterns* (e.g., (Pradel et al., 2010; Raychev et al., 2014)) additionally encode facts such as that a has to be called before b, and b before c. (3) *partial-order patterns* (e.g., (Nguyen et al., 2012)) are modelled as graphs and can encode e.g., that a must be called first, but how b or c are called afterwards is irrelevant.

However, so far, we lack systematic studies of the tradeoffs between the different types of patterns in representing source code in practice. A comparison of different pattern types with regards to some predefined metrics is challenging, because each approach in the literature uses a different learning technique with configurations specific to its data set (e.g., frequency threshold), a different representation for usage examples and patterns, and might even be specifically tied to a particular programming language or input form (e.g., source code vs. bytecode).

In this paper, we address this challenge and present, to the best of our knowledge, the first empirical comparison of API pattern types to investigate their effectiveness in representing API usages in the wild. The different pattern types we compare, consider constraints of different nature between method calls, and thus understanding what exactly they are able to mine

Çergani, E., Proksch, S., Nadi, S. and Mezini, M.
Investigating Order Information in API-Usage Patterns: A Benchmark and Empirical Study.
DOI: 10.5220/0006839000570068
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 57-68
ISBN: 978-989-758-320-9

in a concrete setting constitutes an interesting and relevant subject in many software engineering applications (e.g. code recommendation or misuse detection). To provide a fair setting, we use a common data set of 360 open-source Github C# repositories with over 68*M* lines of code (Proksch et al., 2016) and adopt an established mining algorithm that can be customized to mine all three types of patterns, episode mining (Achar et al., 2012). *Episode mining* is a well-known machine learning technique used to discover partially ordered sets of *events* from a stream, called *episodes* (*patterns* in our terminology). In our setting, *events* are method declarations or invocations (cf. Section 3.2). We can mine all three pattern types by adjusting certain parameters of the episode mining algorithm. With this experimental setup in place, we can produce sequential, partial and no-order patterns using the same mining algorithm and same data set. Our experimental setup is publicly available as a benchmark[1], and can be used by other researchers to perform similar empirical studies.

In this first study, we compare pattern types in terms of three metrics: (1) *Expressiveness* quantifies the richness of the language corresponding to a pattern type whose grammar rules are the mined patterns. We measure expressiveness as the number of words (i.e., derived sequences of method calls) in the language. This measure indicates how well the mined patterns abstract over the variety of concrete API usages observed in source code. Conceptually, one would expect that less structure patterns encode a richer language. The question is, though, to what extent do the differences in expressiveness between pattern types materialize in *the wild*. (2) *Consistency* quantifies the extent to which the words in the language defined by the mined patterns are actually found in the code. This is to judge how truthful the mined API usage patterns represent actual API usage constraints implicitly encoded in source code. From a practical perspective, this metric gives us insights about the relevance of the order information encoded in sequential and partial-order patterns. (3) *Generalizability* measures whether the usages a pattern encodes are specific to a single code context or if they generalize to multiple contexts. In language terminology, this metric indicates whether the learned model is applicable across domains/projects or whether we learn domain-specific languages (models). This is important to understand the applicability of the information encoded in the learned patterns.

The contributions of this paper are as follows:

1. We identify a general episode mining algorithm to fairly compare different pattern types and adapt it to the domain of mining code patterns.
2. We define three metrics on which we base on the comparison between the different pattern types: expressiveness, consistency and generalizability.
3. We perform an empirical study that compares the three pattern types based on the defined metrics. The implications we find from our results help in building better applications based on API usages.
4. We provide a public benchmark that can be used by other researchers to evaluate additional metrics for API usage pattern types.

[1]http://www.st.informatik.tu-darmstadt.de/artifacts/patternTypes/

2 RELATED WORK

Here, we present existing API usage mining techniques and representations, and discuss other studies that have investigated API usages in practice.

2.1 API Usage Representations

API usage representations can be divided into three types: *no-order*, *sequential-order*, and *partial-order*.

No-Order Patterns. The simplest form of learning API usage patterns is to look at frequent co-occurrences of code elements, while ignoring the order these code elements occur in. *Frequent item-set mining* is an example in this category and variations of it have been commonly used (Michail, 2000; Negara et al., 2014; Nguyen et al., 2016).

Sequential-order Patterns. To take code semantics into account, many API usage representations consider order information. For example, calling the constructor of an API type must happen before calling any of its methods. The patterns mined by *sequence mining* encode strict sequential order between code elements in a pattern. Existing approaches are based on, but not limited to, using information from the API's source code (Acharya and Xie, 2009; Wasylkowski et al., 2007), API documentation (Zhong et al., 2009b), program control-flow structure (Ramanathan et al., 2007), and program execution traces (Gabel and Su, 2008; Pradel et al., 2010). Statistical models have also been used to predict the next code element (e.g. method call), given a current context (e.g., sequences of already seen method calls). Examples include n-gram language models (Raychev et al., 2014) or statistical generative models (Pham et al., 2016). Additionally after identifying sequences, some techniques rely on clustering to build pattern abstractions (Wang et al., 2013; Buse and Weimer, 2012; Zhong et al., 2009a).

Partial-order Patterns. This pattern type allows more flexibility in representing code semantics, e.g., that code elements b and c must occur after code element a, but that their order (b before or after c) is not relevant. *Graph-based* techniques like GraLan (Nguyen and Nguyen, 2015), GraPacc (Nguyen et al., 2012), and JSMiner (Nguyen et al., 2014) represent source code in a *graph* to identify frequent sub-graph patterns. *Automata-based* techniques or Finite State Machine (FSM) represent code as a set of states (e.g. method calls) and a transition function between the states. The framework by (Acharya et al., 2007) extract API usage patterns directly from client code. This framework is based on FSMs for generating execution traces along different program paths. In their terminology, partial-order expresses choices between alternative code elements. In our terminology, a partial-order pattern includes strict and/or unordered pairs of code elements.

2.2 Empirical Studies of API Usages

Researchers have extracted API usages through mining software repositories and studied the characteristics of these usages or used them in various applications. Usage patterns are explored in (Ma et al., 2006) from the Java Standard API with an early version of the Qualitas Corpus which contains 39 open source Java applications. A study on a larger corpus (5,000 projects) on usages of both core Java and third-party API libraries is performed in (Qiu et al., 2016). The diversity of API usages in object-oriented software is empirically analyzed in (Mendez et al., 2013). In their context, diversity is defined as the different statically observable combinations of method calls on the same project. Multiple dimensions of API usages are explored in (De Roover et al., 2013), such as the scope of projects and APIs, the metrics of API usages (e.g., number of project classes extending API classes), the API's metadata, and project versus API-centric views.

The empirical study on API usages presented in (Zhong and Mei, 2018), focuses on how different types of APIs are used. Our work is mainly concerned with API patterns instead of single usages. Most previous work focuses on comparing one learning technique with other learning techniques that mine the same pattern type. For example, the framework presented in (Pradel et al., 2010) is used to evaluate three mining approaches that learn all sequences of API method calls. Instead, we focus on understanding the trade-offs between different pattern types.

The work in (Robillard et al., 2013) provides a more comprehensive survey on API property inference and discusses over 60 techniques developed for mining frequent API usage patterns. Overall, existing studies focus on different aspects of API usages, but do not analyze the differences between API usage pattern types. Our work fills this gap and investigates the trade-offs between different API usage pattern types in practice with respect to three metrics: expressiveness, consistency, and generalizability.

3 EPISODE MINING FOR API PATTERNS

We briefly overview the episode mining algorithm and then explain how we use it to mine patterns from open-source C# GitHub repositories, in three steps: (a) generate an event stream by transforming source-code into a stream of events, (b) apply episode mining algorithm to mine API usage patterns, and (c) filter the resulting partial-order patterns.

3.1 Episode Mining Algorithm

To support the detection of sequential-order, partial-order, and no-order patterns in source code, we use the episode mining algorithm (Achar et al., 2012) for the following reasons. *First*, it facilitates the comparison of different pattern types, since it provides one configuration parameter for each type. The other option would be to use different learning algorithms, one per pattern type. In this case, ensuring the same baseline for the empirical comparisons will be difficult, since each algorithm might use different configurations and input formats. *Second*, it is a general purpose machine learning algorithm, which has performed well in other applications: text mining (Achar and Sastry, 2015), positional data (Haase and Brefeld, 2014), multi-neuronal spike data (Achar et al., 2012). *Third*, the implementation of the episode mining algorithm (Achar et al., 2012) is publicly available.

The term *episode* is used to describe a partially ordered set of events. *Frequent episodes* can be found in an event stream through an Apriori-like algorithm (Agrawal et al., 1993). Such an algorithm exploits principles of dynamic programming to combine already frequent episodes into larger ones (Mannila et al., 1997). The algorithm alternates episode candidates generation and counting phases so that infrequent episodes are discarded due to the downward closure lemma (Achar et al., 2012). The counting phase tracks the occurrence of episodes in the event stream using Finite State Automaton (FSA). More specifically, at the *k*-th iteration, the algorithm generates all possible episodes with *k* events by self-joining frequent episodes from the previous iteration consisting of $k-1$ events each. The resulting episodes are episode candidates that need to be verified in the subsequent counting phase. A given episode is *frequent* if it occurs often enough in the event stream. A user-defined *frequency threshold* defines the minimum number of occurrences for an episode to be frequent. An *entropy threshold* determines whether there is sufficient evidence that two events occur in either order or not. All frequent episodes that fulfill the minimum *frequency* and *entropy* threshold are outputted by the algorithm in a given iteration k, and all infrequent episodes are simply discarded. The next iteration begins with generating episodes of size $k+1$. The *entropy threshold* is specific to partial-order patterns. It has a value between 0 and 1, inclusive. A value of 0 means that no order will be mined, resulting in no-order patterns. A value of 1 means a strict ordering of events, resulting in sequential-order patterns. Values between 0 and 1 result in partial-order patterns, with varying levels of strictness. We mine the three pattern types by adjusting the configuration parameter of the episode mining algorithm: *NOC* for No-Order Configuration, *SOC* for Sequential-Order Configuration, and *POC* for Partial-Order Configuration. More details about the algorithm can be found in the work by Achar et al. (Achar et al., 2012).

3.2 Mining API Usage Patterns

Event Stream Generation. In our context, an *event* is any method declaration or method invocation. To transform a repository of source code into the stream representation expected by the episode mining algorithm, we iterate over all source files and traverse each Abstract Syntax Tree (AST) depth-first. Whenever we encounter a method declaration or method invocation node in the AST, we emit a corresponding event to a stream. We use a fully-qualified naming scheme for methods to avoid ambiguous references. The following is how we deal with the two types of nodes we are interested in:

- Method Invocation is the fundamental information that represents an API usage, for which we want to learn patterns. While a resolved AST might point to a concrete method declaration, we generalize this reference to the method that has originally introduced the signature of the referenced method, i.e., a method that was originally declared in an interface or an abstract base class. The reason is that the original declaration defines the *contract* that all derived classes should adhere to, according to *Liskov's substitution principle* (Martin, 2003). Assuming that this principle is universally followed, we can reduce noise in the dataset by storing the original reference.
- Method Declarations represent the start of an enclosing method context that groups the contained method calls. We emit two different kind of events for the encountered method declaration. *Super Context:* If a method overrides another one, we include a reference to the overridden method, i.e., the encountered method overrides a method in an abstract base class. This serves as context information that might be important for the meaning of a pattern. *First Context:* Following the same reasoning as for super context, we include a reference to the method that was declared in an interface that originally introduced the current method signature, which could be further up the type hierarchy of the current class.

We apply heuristics to optimize the event stream generation. (1) We filter duplicated source code, e.g., projects that include the same source files in multiple solutions or that add their references through nested submodules in the version control system. (2) We ignore auto-generated source code (e.g., UI classes generated from XML templates), since they do not reflect human written code. (3) We ignore methods of project-specific APIs (i.e., declared within the same project) to avoid learning project-specific patterns. Our goal is to learn general patterns that have the potential to be re-used across contexts. (4) We ignore references in the data set that point to unresolved types or type elements. These cases indicate transformation errors of the original dataset, that were caused by -for example- an incomplete class path. (5) We do not process empty methods, nor include their method declarations in the event stream.

Learning API Usage Patterns. We feed the generated event stream to the episode mining algorithm after fixing the threshold values: frequency and entropy (as evaluated in Section 4.2).

Filtering Partial-order Patterns. While *SOC* and *NOC* generate episode candidates that are either sequences or sets of events respectively, *POC* might generate episode candidates from all three types, since it contains the sequential and no-order types as special cases. In case all the episode candidates in *POC* are considered frequent episodes during the counting phase, then all of them are outputted by the algorithm. This implies that in every iteration (i.e, pattern size), *POC* might output redundant patterns containing the same set of events but differ in the order information. For illustration, assume that *POC* generates episode candidates in iteration 3 by combing the following patterns from iteration 2: $a \rightarrow b$ and $a \rightarrow c$. The episode candidates in iteration 3 will be: $a \rightarrow b \rightarrow c$ and $a \rightarrow c \rightarrow b$ as sequences, and *a* $\to$ *(b, c)* as partial-order, all possible orderings between the two newly connected events *b* and *c*. The partial-order episode *a* $\to$ *(b, c)* represents both *a* $\to$ *b* $\to$ *c* and *a* $\to$ *c* $\to$ *b*. However, if all three episode candidates turn out to be frequent in the subsequent counting phase, the two other sequences will also be carried over to the next iteration. These redundant patterns are meaningless for source code representation though and we filter them out in each iteration.

4 EVALUATION SETUP

This section describes the data set we use, presents the analyses of the frequency and entropy thresholds used with the episode mining algorithm, and defines the metrics for patterns comparison.

4.1 Data Set

We use an established dataset that consists of a curated collection of 2,857 C# solutions extracted from 360 GitHub repositories (Proksch et al., 2016) with a total of 68*M* lines of source code covering a wide range of applications and project sizes that provide many examples for API usages. The data set uses a specialized AST-like representation of source code with fully-qualified type references and elements. This relieves us from the burden of compiling it to get resolved typing information and makes it easier to transform the source code into the event stream.[2]

We find 138*K* type declarations in the dataset that extend a base class or implement an interface. These type declarations contain 610*K* method declarations. Out of these, 50*K* (first context plus super context) override or implement a method declaration introduced in a dependency. The same dependency can be used in other projects, so focusing on these reusable methods provides valuable context information for the API usage. We find 2*M* method invocations across all method bodies of the data set.

4.2 Frequency and Entropy Thresholds

The episode mining algorithm uses two thresholds: *frequency* and *entropy*. The threshold values directly impact the number of patterns learned: higher threshold values means stronger evidence in the source code that a given pattern occurs. In this section, we empirically evaluate the effects of the threshold values on the number of patterns learned by the three

[2]We use the visitors in the dataset for the transformation.

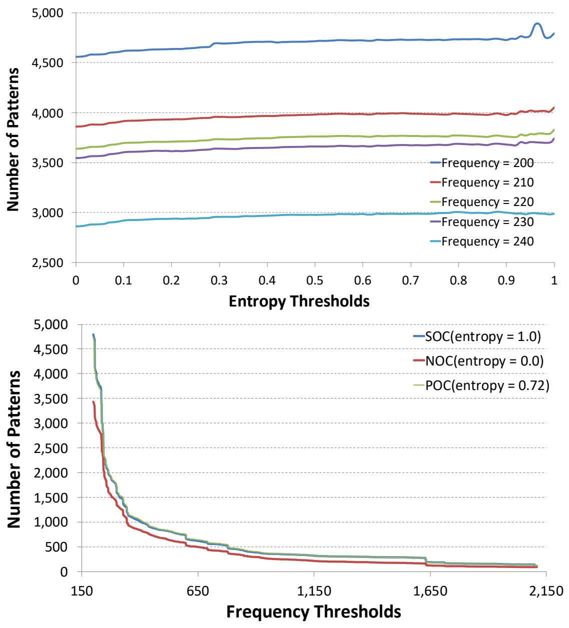

Figure 1: Frequency and entropy threshold analyses.

configurations (*NOC*, *SOC*, *POC*), and select the ones to use for the empirical evaluations presented in Section 5.

Entropy Threshold. Since this threshold is specific to *POC*, we first focus on analyzing the number of patterns learned by *POC* for different entropy and frequency thresholds. Our analyses reveal an increasing number of patterns learned for different entropy thresholds at every frequency level. This is expected, since for entropy values near to 0.0, the algorithm learns mainly unordered sets of events that abstract over several usages. On the other hand, for entropy values near to 1.0 the algorithm learns mainly sequences of events, one for each frequent sequence. For simplicity, Figure 1a shows only a few frequency levels, but similar curves are produced in other frequency levels as well. We observe that for every examined frequency level, *POC* learns a fairly stable number of patterns in the entropy segment of $[0.55, 0.75]$. A stable number of patterns for different threshold values, means that the patterns are not much affected by small fluctuations of the threshold values, making them more preferable compared to an unstable set of patterns that are easily affected by small changes in the threshold values. Our data analyses within this segment reveals that the minimal variation in number of patterns occur for values of $0.71 - 0.72$. Hence, we use the entropy threshold of 0.72 in our next analysis for the frequency threshold and in our empirical evaluations in Section 5.

Frequency Threshold. Our analyses in Figure 1b show that *SOC* and *POC* learn comparable number of patterns for different frequency values, while *NOC* learns less patterns in every frequency level compared to the two others. This is due to the order information: while *SOC* and *POC* may learn multiple patterns for the same set of events, *NOC* simplifies to a single pattern. We select a frequency value that gives a good trade-off between the total number of patterns learned per configuration and comparable number of patterns learned across configurations. Our analyses reveal that this is achieved at the frequency threshold of 345, which we use in the rest of our evaluations. A comparable number of patterns across configurations avoids bias towards one configuration.

4.3 Metrics for Pattern Comparison

We define the following metrics to quantify different properties of the mined patterns in our experiments.

Expressiveness. Using a formal language terminology, an API usage pattern can be seen as a *grammar rule* of a language over an *alphabet* of method declaration/invocation (events). The more words the sublanguage it defines has, the more expressive a pattern is. A *sequential-order* pattern ($a \to b \to c$) when seen as a grammar rule defines a language with a single word $\{abc\}$. A *partial-order* pattern ($a \to (b,c)$) defines a language with two words, $\{abc, acb\}$. A *no-order* pattern (a,b,c) defines a language with six words $\{abc, acb, bac, bca, cab, cba\}$. The expressiveness of a pattern type is determined by the number of patterns (grammar rules) it defines, and how well these patterns abstract over the variety of concrete API usages observed in source code.

To investigate how the three configurations (*SOC*, *POC*, and *NOC*) compare to each other in terms of expressiveness, we calculate three metrics for each configuration pair (c1, c2): (a) exact(c1,c2) is the number of patterns that are exactly the same in c1 and c2; (b) subsumed(c1,c2) = (x,y) is a pair that represents the number of patterns x learned by c1 that subsume y patterns learned by c2. We say that a pattern $p1$ subsumes a pattern $p2$ iff they relate the same set of events and all words defined by $p2$ are also defined by $p1$, e.g. the grammar rule of a no-order pattern (a,b,c) subsumes both the grammar rules $(a \to (b,c))$ and $(a \to b \to c)$ from the partial and sequential-order patterns respectively; (c) new(c1,c2) is the number of patterns learned by c1 that include events for which c2 does not learn any pattern.

Consistency. The three pattern types differ in the extent to which they preserve code structure. While *no-order* patterns cannot represent any structure, *sequential-order* patterns can encode an absolute order of events, and *partial-order* patterns can even represent complex control flow that is imposed by control structures like if. We establish the *consistency* metric as a way to quantify how important the order information encoded by sequential-order and partial-order patterns is in practice. The metric takes values in $]0.0, 1.0]$, and for a given pattern p is defined as:

$$consistency(p) = \frac{Occs(p)}{OccsSet(p)} \quad (1)$$

where $Occs(p)$ is the number occurrences of p, and $OccsSet(p)$ is the number of co-occurrence of events in p regardless of their order. A high consistency emphasizes the importance of the encoded order. A low consistency means that in most cases, the respective code elements occur in an order different to the one encoded in the pattern, suggesting that the structural information encoded by the pattern is irrelevant.

Generalizability. Finding instances of a pattern in multiple contexts indicates that the pattern represents an abstraction over a set of similar API usages, e.g., used by multiple developers. On the other hand, a very *local* pattern might suggest that it does not generalize beyond a specific context, e.g., it might only be used by a specific developer. To quantify the *generalizability* of a pattern, we count the number of contexts in which we can observe it at two different levels of granularity that complement each other: (a) The *method declaration* level measures whether instances of a pattern are found within a single method declaration (the latter refers to the highest declaration in the type hierarchy that originally introduced the current method signature) or across method declarations (method-specific versus cross-method pattern). (b) The *code repository* level measures whether instances of a pattern are found in one or in multiple repositories (repository-specific versus cross-repository pattern). Knowledge about the generalizability of patterns is important for judging the versatility of the pattern in later applications.

5 STUDY RESULTS

This section presents the results of our empirical study. All experiments are performed with a frequency threshold of 345. For *POC*, we use an entropy threshold of 0.72 (cf. Section 4.2). First, we show statistics about the learned patterns, and then study them along the dimensions presented in Section 4.3.

5.1 Pattern Statistics

Here we analyze the learned patterns in terms of their size and number of API types they encode.

Pattern size refers to the number of events in a pattern. Our approach learns patterns with up to 7 events in each configuration. The number of patterns learned decreases for larger pattern sizes with the same ratio in each configuration. Almost all mined patterns (97%) involve 5 events or less. The result matches the intuition that it is less probable that many developers write large code snippets in exactly the same way.

API types within a pattern reflects the number of API types a pattern encodes interactions for. In all the patterns learned, 75% involve interactions between events from multiple API types (across configurations). Only 28% of the patterns with 2 - 4 events involve interactions between events from a single API type. All patterns with 5 or more events involve multiple API types. The maximum number of API types involved within a pattern is 5 types, where patterns involving two API types make the majority (40%).

5.2 Expressiveness

Table 1 shows the expressiveness metric results. For each configuration pair (c1,c2), *Total* shows the total number of patterns learned by (c1,c2) respectively.

POC vs. _SOC_. These configurations learn 858 equal patterns, which implies that out of 1,234 patterns learned by *POC*, 70% are sequences and only 30% of them include partial-order between events.

Observation 5.1
Most of the API usage patterns define in the wild strict-order between events (70%), while the other 30% abstract over different API usage variants.

Furthermore, subsumed(*POC*, *SOC*) is (260;346), i.e., 260 partial-order patterns learned by *POC* subsume 346 sequences learned by *SOC*. The 260 partial-order patterns encode 572 different sequences, i.e., the 346 sequences mined by *SOC* plus 226 others. Recall that multiple sequential-order patterns can be represented by a single partial-order pattern.

Finally, new(*POC*, *SOC*) is 116, meaning that for the events included in 116 partial-order patterns, there are no sequences learned by *SOC*. The 116 partial-order patterns encode 308 sequences of events that individually do not occur often enough in source code. For this reason, *SOC* does not mine them. On the other hand, *POC* represents different variants of sequences for the same set of events in a single pattern, which increases the partial-order pattern occurrence and makes it match the frequency threshold.

Table 1: Expressiveness results per configuration pair.

	(*POC*, *SOC*)	(*NOC*, *POC*)	(*NOC*, *SOC*)
exact	858	248	0
subsumed	(260;346)	(716;986)	(853;1204)
new	116	17	128
Total	(1,234;1,204)	(981; 1,234)	(981;1,204)

From these results, we can conclude that all patterns learned by *POC* represent a superset of the patterns learned by *SOC*.

Observation 5.2
The API usage specifications encoded by partial-order patterns fully represent the specifications encoded by sequential-order patterns. Furthermore, they learn 116 additional patterns of events for which sequence mining cannot learn any sequence.

NOC vs. _POC_. As shown in Table 1, exact(*NOC*, *POC*) = 248, which means that 20% of the patterns learned by *POC* are exactly the same as the ones learned by *NOC*. Recall that no-order patterns are mined in *POC* when the involved events occur often enough in either order.

Observation 5.3
In 20% of the cases, partial-order patterns encode events that occur in either order in the wild.

Furthermore, subsumed(*NOC*, *POC*) is (716; 986), i.e., 716 no-order patterns learned by *NOC* subsume 986 patterns learned by *POC*. Note that one no-order pattern simplifies several partial-order patterns that misses order information.

Finally, new(*NOC*, *POC*) is 17, i.e.,17 patterns learned by *NOC* include events for which *POC* does not learn any pattern. These patterns are missed by *POC* because either: (a) none of the sequences between the events occur frequently enough, recall that sequences are a special case of partial-order patterns, and/or (b) there is not enough evidence in the source code that events occur frequently enough in either order (specified by entropy threshold).

From these results we can conclude that no-order patterns represent a superset of partial-order patterns.

NOC vs. _SOC_. Table 1 shows that *NOC* and *SOC* learn 0 equal patterns, which is obviously the case, since *NOC* learns only set of events and *SOC* learns only strict-order sequences, i.e., there cannot be any overlap between the patterns learned by these two configurations. We find that subsumed(*NOC*, *SOC*) is

(853; 1,204). In other words, all sequential-order patterns can be subsumed by 853 no-order patterns. Note that multiple sequential-order patterns can be simplified into a single no-order pattern by removing order constraints.

Finally, new(*NOC*, *SOC*) is 128, i.e., for 128 patterns learned by *NOC* there are no sequences mined by *SOC*. None of the sequences between these events occurs frequently enough in the source code.

Observation 5.4
No-order patterns match all sequential-order patterns; furthermore, the no-order configuration learns 128 additional patterns for which sequential-order configuration could not learn any sequences.

Analysis of the Results. To recap, sequence mining misses sequences of events which are captured by partial and no-order patterns. To understand what code structures they represent, we explored mined patterns and found examples that explain this phenomenon in the source code of Graphical User Interfaces (GUI). Using a GUI component typically requires to call its constructor first, but the order in which properties like color or size are configured is irrelevant. A miner thus finds many UI code examples with high variation and low support of each individual example. This reveals two disadvantages of the sequential-order miners. *First,* if the individual support for each variant of the GUI component usage is high enough, then redundant patterns will be identified, one sequence for each variant. *Second,* if the target threshold is not met by one or more sequence variants, the corresponding sequence pattern will be missed. In the same situation, each variant would count as support for patterns with more abstract representation such as partial and no-order, which thus may pass the threshold more easily. When compared with each other, partial-order patterns can preserve order information, which is missed by no-order patterns.

5.3 Consistency

Based on the results in 5.2, one may conclude that no-order patterns define a richer language compared to the other two types. The question raises: Why should one use expensive mining approaches (sequence or partial mining), if we can learn a richer language from source code using less computationally expensive mining approaches such as frequent item-set mining? However, this would be a valid conclusion, only if the words in the language mined by NOC are valid, i.e., the order between events in a pattern does not really matter. To analyze this, we investigate the consistency of the mined sequential and partial-order patterns with co-occurrences of events in code.

Our results reveal high consistency in sequential (avg. 0.9) and partial-order patterns (avg. 0.96). This suggests that order information encoded in both sequential and partial-order patterns is crucial for the correct co-occurrences of events in the wild, and simplifying them into no-order patterns will result in losing important order information between events.

Observation 5.5
Partial and sequential-order mining learn important order information regarding co-occurrences of events within a pattern.

5.4 Generalizability

In this section, we present the generalizability metric results on two granularity levels as explained in Section 4.3: method declaration and code repository.
Method Declaration. Our results empirically show that most of the patterns learned (98%) by each configuration, are used across method declarations. If a pattern occurs across method declarations, it means that it generalizes to different implementation tasks.

Observation 5.6
Most of the patterns learned find applicability to a large variety of implementation tasks.

Next we analyze if the patterns learned are used by multiple developers, or if they represent specific coding styles for a given repository and its developers.
Code Repository. Table 2 shows our results for different configurations and pattern sizes. The column *Patterns* shows the total number of patterns, and the absolute number and percentage of general patterns learned by each configuration. The next columns show the same information as *Patterns*, but for different pattern sizes, where the last column (6+ events) shows the information for patterns that have more then 6 events.

Our results show that the patterns learned by *POC* and *SOC* have almost the same percentage of generalizability (48% vs. 47%), regardless of their size. This means that more than half the patterns mined by each configuration are learned from API usages from the same repository. While such repository-specific patterns are useful to the developers of that particular repository, they may reflect a very specific way of using certain API types, which may not be useful to a general set of developers.

As the table shows, *NOC* learns slightly more general patterns (58%). However, recall that these more generalizable patterns come at the cost of missing order information between events.

Table 2: Code repository generalizability level for different configurations and pattern sizes.

	Patterns		2 events		3 events		4 events		5 events		6+ events	
Config	Total	General	Total	General	Total	General	Total	General	Total	General	Total	General
POC	1,234	594 (48%)	573	472 (82%)	283	106 (38%)	212	15 (7%)	122	1 (1%)	44	0 (0%)
SOC	1,204	561 (47%)	562	458 (82%)	270	92 (34%)	206	10 (5%)	122	1 (1%)	44	0 (0%)
NOC	981	572 (58%)	528	445 (84%)	226	108 (48%)	132	17 (13%)	70	2 (3%)	25	0 (0%)

Observation 5.7
No-order patterns tend to be more generalizable (58%) compared to sequential and partial-order patterns (47% and 48%), which tend to be over-specified due to the order constraints they encode.

We analyzed the patterns learned exclusively by *POC* (recall Table 1) and found that 114 out of 116 patterns are general patterns used across repositories. To find out why most of the patterns learned exclusively by *POC* are general patterns, we check if there is any relation between generalizability and pattern-order. We find that strict-order patterns (exact(*POC*, *SOC*)) are less generalizable (37%) compared to patterns that contain partial-order between events (*subsumed* - 62%, and *new* - 98%). This confirms our hypothesis that there is a relation between generalizability and pattern-order. Furthermore, most of the patterns (90%) learned exclusively by *POC* include method calls only from the standard library, which further explains their re-usability across repositories.

Table 2 shows that across configurations, the percentage of general patterns learned is higher for smaller patterns, and significantly decreases for bigger patterns. Furthermore, for patterns with 6-events and more, we learn only repository-specific patterns. Specifically, around 70% of general patterns (independent of the configuration) are 2 and 3-event patterns. Most of the patterns with 4-events or more are repository-specific patterns. This makes sense since the probability that multiple developers with different coding styles and different application domains writing a similar and long piece of code is very low.

Observation 5.8
Small code patterns of 2 and 3 events are more generalizable compared to larger code patterns of 4 or more events that mainly encode constraints of API usages from a single repository.

We further analyzed the repository-specific patterns and found that 93% of them are learned from testing code, and they include API types that refer to an old version of a common assembly that is used in no other repository. Filtering out testing code may help mining algorithms learn only general patterns. An empirical validation of this hypothesis, however, needs to be performed in the future.

Remark: For the sake of completeness, we experimented with other threshold values (frequency and entropy), and analyzed the generalizability of the patterns across repositories. The results we received did not show higher generalizability ratios in neither of the pattern types, compared to the ones presented above. This confirms the correctness of the threshold values selected as presented in section 4.2.

6 IMPLICATIONS

Based on the pattern statistics (Section 5.1) and results in Section 5, we derive the following:

Implication 1 (derived from Section 5.1). Mining techniques based on frequency occurrence of source code in code bases are unlikely to learn large code patterns (more than 7 method calls using our concrete parameters), since it is less probable that developers write large code snippets exactly in the same way. If the main goal is to learn large code patterns, then other techniques need to be considered.

Implication 2 (derived from Section 5.1). Code analyses techniques should consider interactions between objects of different API types, while extracting facts from source code. Even though these analyses are expensive since data-flow dependencies need to be considered, they are important in mining relevant patterns from source code.

Implication 3 (derived from Observations 5.1 and 5.5). While covering a good amount of usages seen in source code, sequential-order mining may lead to false positives in applications such as misuse detection. For example, if the pattern is $a \rightarrow (b,c)$, but a strict-order pattern has only learned $a \rightarrow b \rightarrow c$ and the code written by the developer is $a \rightarrow c \rightarrow b$. On the other hand, while no-order mining might seem to learn a larger variety of API usages in source code, it might result in false negatives in such applications. Following the same example, the developer might have written $b \rightarrow a \rightarrow c$, and a no-order pattern cannot detect that b and c should occur strictly after a. We can conclude that, partial-order mining learns better API usage patterns for such applications.

Implication 4 (derived from Observation 5.2). Partial-order mining might be more appropriate for learning API usage patterns in applications such as code recommendation since multiple sequences can be represented by a single partial-order pattern, decreasing the total number of patterns that need to be part of the model. In sequence mining, multiple patterns need to be recommended to the developer for the same set of events and might even risk missing valid sequences if they do not occur frequently enough in the training source code.

Implication 5 (derived from Observation 5.5). Before deciding which mining approach to use in a specific application, developers need to know their trade-offs in terms of order information and computation complexity. Sequential and partial-order mining are computationally expensive approaches but learn important order information about the co-occurrence of events in a pattern, while no-order mining approaches do not require expensive computations but on the other hand do not learn any order information about the co-occurrence of events in a pattern.

Implication 6 (derived from Observation 5.8). If the main goal is to learn large code patterns (4 - 7 events), then recommenders should focus on a repository-specific mining approach and produce catered recommendations to the repository's developers. However, if the goal is to learn general patterns that can be used by many developers, then researchers should know that they might end up mining small patterns (2 and 3 events).

7 THREATS TO VALIDITY

Internal Validity. We generate the event stream based on static analyses, not on dynamic execution traces. Even though this may not represent valid execution traces, it does represent how the code is written by developers. In this paper, we focus on learning code patterns to represent source code as it is written in code editors. Also, our event stream considers only intra-procedural analysis since we are interested to learn patterns that occur within methods. Using inter-procedural analysis might affect our results.

The episode mining algorithm learns only *injective* episodes, where all events are distinct, i.e., the algorithm does not handle multiple occurrences of the same event in a pattern. For example, method invocations: IEnumerator.MoveNext() or StringBuilder.Append() are usually called multiple times in the code. The patterns we learn contain a single instance of such events. While this is a limitation, it is also an advantage in terms of pattern generalizability. Specifically, the mined pattern would not have a strict number of occurrences that would lead to mismatches between it and another valid code snippet that has a different number of occurrences.

The algorithm relies on user-defined parameters: frequency-threshold, entropy-threshold. While the configuration parameter depends on the type of patterns one is interested in, deciding on adequate frequency and entropy thresholds is not an easy task, which affect the results. We mitigate this threat by empirically evaluating the thresholds and choosing the best combination of frequency and entropy thresholds for the given data set (cf. section 4.2).

The episode mining algorithm is available only in a sequential (non-parallelized) implementation, hence is inefficient. However, this paper does not advocate using episode mining per se, but rather uses it as a baseline for comparing different configurations. This limitation can be improved by parallelizing the algorithm's implementation.

External Validity. In this paper, we do not learn patterns for project-specific API types. Extracting code patterns for project-specific API types can still be achieved using the episode-mining algorithm we use. Comparing project-specific patterns between different types of projects is an interesting task for future work.

We learn code patterns only for *method declarations* and *invocations*, excluding all other code structures such as *loops*, *conditions*, *exceptions* etc. This is because the focus of this paper is on comparing different code pattern types (sequential, partial, and no-order), instead of specifically learning complex patterns that include all code structures. Since learning code patterns while considering other code structures is important for supporting certain development tasks, we plan to enrich the code patterns that we learn with additional code structures. This requires modifying our event stream generation, which is an engineering task rather than a conceptual limitation.

Finally, we analyze the trade-offs between different pattern types using the same set of code repositories written in the same programming language. We also use a single learning algorithm that we configure to produce different pattern types. We use an established data set of 360 repositories that have over 68*M* lines of source code to ensure that we analyze large amounts of code and different coding styles. However, we cannot generalize our results beyond our current dataset and learning algorithm.

8 CONCLUSIONS

In this paper, we present the first benchmark for analyzing the trade-offs between three pattern types (sequential, partial and no-order) with respect to real code. Our approach consists of three steps: the transformation of source-code into a stream of events, the adaptation of an event mining algorithm to the special context of pattern mining for software engineering, and filtering of the resulting patterns.

Our empirical investigation shows that there are different types of patterns learned in code repositories. While there are tradeoffs between pattern types in terms of *expressiveness*, *consistency* and *generalizability*, they are comparable in terms of the patterns size and number of API types. Our results empirically show that the *sweet spot* are *partial-order* patterns, which are a superset of *sequential-order* patterns, without losing valuable information like *no-order* patterns. Partial-order mining finds additional patterns that are missed by sequence mining, which generalize across repositories. Compared to no-order mining, partial-order learns a smaller percentage of cross-repository patterns (58% vs. 48%), due to the order constraints between events within a pattern. Evaluation results show that all three configurations end-up learning only *repository-specific* patterns for pattern sizes with *6-events* or more. Furthermore, our results empirically show the *consistency* of order information in sequential and partial-order patterns: on average 90% and 96% respectively.

Our findings are useful indications for researchers who work with code patterns in applications such as code recommendation and misuse detection.

ACKNOWLEDGEMENTS

This work has been supported by the European Research Council with grant No. 321217, and by the German Science Foundation (DFG) in the context of the CROSSING Collaborative Research Center (SFB #1119, project E1). The authors want to thank Raajay Viswanathan for the technical support with the episode mining algorithm, and Ulf Brefeld for the useful suggestions on the analyses of the data presented on this paper. The authors take full responsibility for the content of the paper.

REFERENCES

Achar, A., Laxman, S., Viswanathan, R., and Sastry, P. (2012). Discovering injective episodes with general partial orders. *Data Mining and Knowledge Discovery*, pages 67–108.

Achar, A. and Sastry, P. (2015). Statistical significance of episodes with general partial orders. *Information Sciences*, pages 175–200.

Acharya, M. and Xie, T. (2009). Mining API error-handling specifications from source code. In *International Conference on Fundamental Approaches to Software Engineering*, pages 370–384.

Acharya, M., Xie, T., Pei, J., and Xu, J. (2007). Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 25–34.

Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *ACM SIGMOD*, pages 207–216.

Buse, R. P. and Weimer, W. (2012). Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, pages 782–792. IEEE Press.

De Roover, C., Lammel, R., and Pek, E. (2013). Multidimensional exploration of api usage. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 152–161. IEEE.

Gabel, M. and Su, Z. (2008). Javert: fully automatic mining of general temporal properties from dynamic traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 339–349.

Haase, J. and Brefeld, U. (2014). Mining positional data streams. In *International Workshop on New Frontiers in Mining Complex Patterns*, pages 102–116.

Ma, H., Amor, R., and Tempero, E. (2006). Usage patterns of the java standard api. In *Software Engineering Conference, 2006*, pages 342–352.

Mannila, H., Toivonen, H., and Inkeri Verkamo, A. (1997). Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, pages 259–289.

Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

Mendez, D., Baudry, B., and Monperrus, M. (2013). Empirical evidence of large-scale diversity in API usage of object-oriented software. In *Source Code Analysis and Manipulation*, pages 43–52.

Michail, A. (2000). Data mining library reuse patterns using generalized association rules. In *International Conference on Software Engineering*, pages 167–176.

Montandon, J. E., Borges, H., Felix, D., and Valente, M. T. (2013). Documenting APIs with examples: Lessons learned with the APIMiner platform. In *WCRE*, pages 401–408.

Negara, S., Codoban, M., Dig, D., and Johnson, R. E. (2014). Mining fine-grained code changes to detect unknown change patterns. In *International Conference on Software Engineering*, pages 803–813.

Nguyen, A. T., Hilton, M., Codoban, M., Nguyen, H. A., Mast, L., Rademacher, E., Nguyen, T. N., and Dig, D. (2016). Api code recommendation using statistical learning from fine-grained changes. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522.

Nguyen, A. T. and Nguyen, T. N. (2015). Graph-based statistical language model for code. In *International Conference on Software Engineering*, pages 858–868.

Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Tamrawi, A., Nguyen, H. V., Al-Kofahi, J., and Nguyen, T. N. (2012). Graph-based pattern-oriented, context-sensitive source code completion. In *International Conference on Software Engineering*, pages 69–79.

Nguyen, H. V., Nguyen, H. A., Nguyen, A. T., and Nguyen, T. N. (2014). Mining interprocedural, data-oriented usage patterns in javascript web applications. In *International Conference on Software Engineering*, pages 791–802.

Pham, H. V., Vu, P. M., Nguyen, T. T., et al. (2016). Learning API usages from bytecode: a statistical approach. In *International Conference on Software Engineering*, pages 416–427.

Pradel, M., Bichsel, P., and Gross, T. R. (2010). A framework for the evaluation of specification miners based on finite state machines. In *IEEE International Conference on Software Maintenance*, pages 1–10.

Proksch, S., Amann, S., Nadi, S., and Mezini, M. (2016). A dataset of simplified syntax trees for c#. In *International Conference on Mining Software Repositories*, pages 476–479.

Qiu, D., Li, B., and Leung, H. (2016). Understanding the api usage in java. *Information and Software Technology*, pages 81–100.

Ramanathan, M. K., Grama, A., and Jagannathan, S. (2007). Path-sensitive inference of function precedence protocols. In *International Conference on Software Engineering*, pages 240–250.

Raychev, V., Vechev, M., and Yahav, E. (2014). Code completion with statistical language models. In *ACM SIGPLAN Notices*, pages 419–428.

Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M., and Ratchford, T. (2013). Automated API property inference techniques. *IEEE Transactions on Software Engineering*, pages 613–637.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., and Zhang, D. (2013). Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 319–328. IEEE Press.

Wasylkowski, A., Zeller, A., and Lindig, C. (2007). Detecting object usage anomalies. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 35–44.

Zhong, H. and Mei, H. (2018). An empirical study on API usages. *IEEE Transaction on Software Engineering*.

Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009a). MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343.

Zhong, H., Zhang, L., Xie, T., and Mei, H. (2009b). Inferring resource specifications from natural language API documentation. In *International Conference on Automated Software Engineering*, pages 307–318.