Towards Assisting Developers in API Usage by Automated Recovery of Complex Temporal Patterns

Mohamed Aymen Saied[a], Erick Raelijohn[b], Edouard Batot[b], Michalis Famelis[b], Houari Sahraoui[b,*]

[a]*Université du Québec à Rimouski, Canada*
[b]*Université de Montréal, Canada*

Abstract

Context: Despite the many advantages, the use of external libraries through their APIs remains difficult because of the usage patterns and constraints that are hidden or not properly documented. Existing work provides different techniques to recover API usage patterns from client programs in order to help developers use those libraries. However, most of these techniques produce patterns that generally do not involve temporal properties.
Objective: In this paper, we discuss the problem of temporal usage patterns recovery and propose an algorithm to solve it. We also discuss how the obtained patterns can be used at different stages of client development.
Method: We address the recovery of temporal API usage patterns as an optimization problem and solve it using a genetic-programming algorithm.
Results: Our evaluation on different APIs shows that the proposed algorithm allows to derive non-trivial temporal usage that are useful and generalizable to new API clients. *Conclusion:* Recovering API usage temporal patterns helps client developers to use APIs in an appropriate way. In addition to potentially improve productivity, such patterns also helps preventing errors that result from an incorrect use of the APIs.

1. Introduction

In modern software development, achieving any meaningful task of non-trivial complexity means that developers must reuse software from multiple sources in the form of APIs, libraries, and services. Such libraries usually require that client applications obey assumed constraints and usage patterns. Such constraints can be a barrier to adoption by developers, as learning them is time consuming and tedious, depending heavily on the quality of documentation. To make matters worse, such directives are generally not well-documented [23, 31]. For example, they may touch many methods/classes, whereas documentation such as Javadoc tends to be unit based (per class, per method).

Specification mining is one way to address these problems. Mined specification patterns can be used to complement the documentation of libraries. This can be done by, for example, providing developers with typical usage scenarios or by integrating the mined patterns into IDEs to provide on the fly recommendations.

Recently, much research effort has been dedicated to the identification of sequential API usage patterns (ordered sets of co-used methods) [35] based on development activities [2] and execution traces [11, 22, 30]. Other research contributions targeted unordered API usage patterns [9, 26, 27, 28, 29]. Given a set of client programs that use a library of interest, existing techniques identify API usage patterns that are recurrent in them. However, leveraging mined patterns to help developers ensure that the software remains correct while features are added or removed remains a challenge as the usage patterns inferred with such techniques tend to be simple, numerous, and with high degrees of redundancy.

However, mining temporal aspects of such constraints (i.e., latent temporal properties of APIs) is still an open question. Some work has been done on mining temporal specifications from libraries in the form of automata [17] or rules [9, 18]. Others have attempted to uncover latent behaviour in UML models [10]. On the one hand, a mined automaton expresses a global picture of library specification, but the graph may be very complex and not practical. On the other hand, mined rules generally consist of two events, i.e., two method calls, which limits their ability to express complex temporal properties. This is due to the fact that mining approaches generally rely on predetermined templates, such as the property patterns of Dwyer et al. [6]. Thus, only specific classes of constraints can be identified, instead of all the possibilities that a person could build, require, or understand. Further, to the best of our knowledge, the published literature does not address the mining of APIs in particular.

In this paper, we propose to generalize existing approaches for learning temporal API constraints without using predetermined templates. To handle a wider spectrum of constraints, we define a probabilistic ap-

*Corresponding author

proach centered around the use of atomic constraint subexpressions as "building blocks" for a search-based exploration of the space of possible API constraints. To this end, we propose a genetic-programming technique that gradually builds Linear Temporal Logic (LTL) formulas, representing candidate usage patterns, by combining API method calls with logical and temporal operators. The search-space exploration is guided by the conformance of candidate patterns with execution traces of client programs using the targeted API.

We evaluated our approach on eight APIs having a variable number of clients. Our evaluation shows that we obtained patterns with different sizes and complexities. It also evidenced that these patterns are generalizable clients not seen in the learning phase.

This paper extends our previous work [30] that was published in the Genetic and Evolutionary Computation Conference (GECCO) as follows:

a) We provide extensive details about the experimental setup of the pattern mining validation.

b) We re-implemented a baseline approach and compared with our approach through shedding light on the kind of patterns that could be inferred.

c) We introduce Tapir (*Temporal API Recommender*), a tool that allows putting the mined temporal investigate to use within developers' IDEs. Specifically, we envision using Tapir in four contexts: First, before the developer starts writing the API client application code, we envision using Tapir to help augment existing API documentation by translating patterns into structured natural language. Second, Tapir can help when developers write the client-application code. This is done by flagging potential misuses or by refining the code completion suggestions for API calls. Additionally, Tapir can be used at testing time to respectively assess whether the code and the execution traces satisfy the mined patterns. A planned extension to Tapir, will allow us to adapt this for use at compilation time, too.

d) We present real-word running examples to illustrate the different ways to leverage the mined API temporal patterns.

e) We propose a method, implemented in Tapir, to help client application developers to correctly use an API at different development phases using the LTL patterns. Specifically, we propose a method based on the generation and analysis of "pseudo-traces" from complete and partial source code, using static analysis.

The rest of the paper is organized as follows. Section 2 outlines the contours of the problem of automated mining of temporal API usage patterns. Section 3 includes an overview of the related work, and we discuss our vision on how existing approaches can be extended to target a wide spectrum of API usage constraints. The details of our approach and its evaluations are provided respectively in Section 4 and Section 5. In Section 6, we present a research agenda for leveraging mined API usage patterns in practice. We conclude this paper in Section 7.

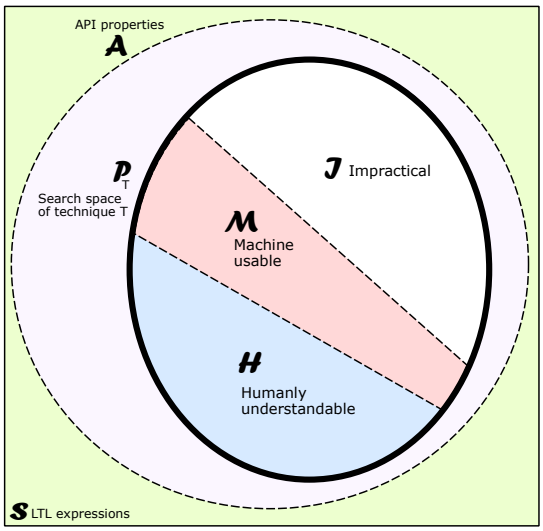

Figure 1: Overview of the search space.

2. Problem Discussion

In this section, we outline the contours of the problem of automated recovery of temporal API usage patterns. In this paper, we focus on recovering such patterns in the form of Linear Temporal Logic (LTL) expressions. We symbolize as $\mathcal{S}$ the space of all well-formed LTL expressions. Given an API, we denote as $\mathcal{A} \subseteq \mathcal{S}$ the set of all LTL expressions that describe its valid usage patterns, *i.e.* LTL expressions involving the API public methods. Ideally, a recovery technique can search all of $\mathcal{A}$ to find useful patterns (for some definition of "useful"). In practice, however, techniques make assumptions that limit the size of the space within which they search. We call such assumptions *search space parameters*. These parameters include limitations on the size and complexity of the candidate patterns, and the predefinition of the pattern templates to consider. Other possible parameters are thresholds on the frequency and accuracy of the candidate patterns on the learning data. Indeed, when a candidate pattern occurs very frequently, it might be the sign of triviality or tautology. Similarly, one may decide to consider only candidate patterns that are never contradicted in the learning data or tolerate a certain degree of falsification. Thus in practice, a given technique T searches within a subspace $\mathcal{P}_T \subseteq \mathcal{A}$ defined by its search space parameters. We call this the *search space* of the technique. When the choice of technique is obvious, we write simply $\mathcal{P}$.

We further define three subregions of $\mathcal{P}$, depending on the "usefulness" of the patterns that can be discovered in each one. The first subregion, symbolized by $\mathcal{H}$, contains all patterns that can be understood by humans with reasonable cognitive effort. These are, for example, patterns that follow the guidelines proposed by Dwyer

et al [6]. Patterns in $\mathcal{H}$ are generally small with low complexity. The second subregion, symbolized by $\mathcal{M}$, contains the patterns that, while not humanly comprehensible, can still be efficiently leveraged by automated techniques to assist humans in the correct usage of APIs. This can take various forms, such as embedding into an IDE a recommender system that suggests the most likely API method to be used next, or detecting uncommon and suspicious API usages. Patterns in $\mathcal{M}$ are large and/or complex. The final subregion, symbolized by $\mathcal{I}$, contains patterns that cannot be practically used either by humans or automated techniques. This patterns are generally too large and too complex, and are generally extremely frequent in the learning data. Obviously $\mathcal{H} \cup \mathcal{M} \cup \mathcal{I} = \mathcal{P}$. An overview of all the different regions of the search space is illustrated in Figure 1.

3. Related Work

In recent years, much research effort has been dedicated to specification mining. For instance, SpecForge [14], synergize many existing finite state automaton based on specification mining algorithms. SpecForge generates a superior FSA from a set of FSAs mined with existing algorithms. SpecForge extracts important constraints that are common across the mined FSAs and combine the extracted constraints into one FSA model. It also uses linear temporal logic to specify ordering constraints satisfied by the FSA. In this paper, we are specifically interested in the mining of API specifications and usage patterns [24]. The inference can generate unordered API usage patterns that describe which API elements ought to be used together without considering the order of usage. However, some approaches, also, take order into account and provide more detailed knowledge about the correct use of an API in the form of sequential or temporal usage patterns [24].

3.1. Unordered Usage Patterns Mining

Existing techniques for mining unordered API usage patterns are valuable to facilitate API understanding and usage.

Zhong et al. [38] developed the MAPO tool for mining API usage patterns. MAPO clusters frequent API method call extracted from code snippets, based on the number of called API methods and textual similarity of class and method names between different snippets.

Zhu et al. [40] propose an approach to mining API usage examples from unit test code. They employed a slicing technique to separate test scenarios into code examples. Then they cluster the similar usage examples for recommendation.

Saied et al. [29] propose a clustering-based approach to mining API usage patterns. Their technique combines a static analysis of the API code (structural and semantic dependencies of API methods) with a dynamic analysis of execution traces of client programs (API methods co-usage).

3.2. Temporal Usage Patterns Mining

As mentioned in Section 2, the space of possible API temporal patterns is very large and potentially infinite. One needs to use heuristics to explore this space. In the following, we discuss the heuristics used and their impact on the patterns that can be found.

A simple heuristic to explore the space of possible patterns is to search for specific pattern templates. This has been done, for example, by Yang *et al.* in the Perracotta approach [37]. In this work, simple two-events patterns are mined from the execution traces. These patterns follow the template *event1 always followed by event2*. Such simple patterns, once mined, can be combined to produce larger patterns. A similar combination strategy was also proposed by Gabel and Su in [8], but with a different formalism to represent the patterns. As stated by Lo *et al.*, such combination methods might miss some multi-event patterns or create ones that are not meaningful [18]. Alternatively, Lo *et al.* generalize the two-event template to a multi-event template, and mine patterns in the form of *whenever a series of events occurs, eventually another series of events will occur*. Uddin et al. [35] also considered the multi-event template. They detect temporal sequential patterns of API use in terms of their time of addition into the source code during the client program development. Due to the volatile nature of the development process, detected patterns may suffer from some imprecision.

The above-mentioned approaches cannot mine complex patterns such as *When event1 occurs, either event2 occurs just after and event3 will never occur after, or event2 will not occur just after and event3 will eventually occur*. Lemieux *et al.* [15] proposed a model-checking-base tool, Texada, to mine such complex patterns. However, the pattern templates used for the search have to be specified beforehand. In other words, it is impossible to find what you are not explicitly looking for. This precludes exploratory searching, which can reveal previously unanticipated patterns. A good metaphor to understand the difference between mining specific pattern templates vs finding unspecified patterns is one of the difference between automated analysis and visualization in program comprehension. In the first, the goal is to get a precise response to a specific question, whereas, in the second case, the goal is to get an abstraction of the system, which can be explored even without a specific question in mind.

3.3. Generalizing the Mining

The goal of our work is to generalize the template-based mining of temporal API usage patterns to a non template-based mining while keeping the search for temporal patterns feasible. Rather than using the template heuristic, we use a meta-heuristic to explore the space of possible patterns, i.e, genetic programming. In this approach, we

progressively construct sets of patterns of arbitrary form and test their accuracy using the Texada tool [15].

3.4. API Pattern Usage

From the early years, teams have been mining software artifacts in the search for useful information to link to documentation and good-practice guidelines. The closest to our study is the work from Cergani *et al.* [3]. In this paper, authors mine software repository to evaluate thresholds between information at different level of order (Sequential, partial order and non ordered). With a specialized AST-like representation, they derive event streams to simulate software usages. They show empirically the efficiency of partial-order information *i.e.* like LTL patterns. The main difference with our work reside in the use of real traces to feed the learning process. Other works attempt to augment API documentation by means of software repository mining. Teams automatically mine API usages from source code with consideration regarding the frequencies of connected subgraphs [39, 4]. Finally, the form in which the result of such investigation are presented to end users (*i.e.* generally developers) varies. Subramanian *et al.* mine bidirectional link between documentation and code snippets (or patterns) derived from source code and produce a HTML report [33]. Sun *et al.* mine software repository to assist the derivation of interfaces [34].

4. Complex temporal API usage patterns mining

Our approach consists in using genetic programming, an evolutionary method, to mine temporal patterns from execution traces. Genetic programming is a powerful search method inspired by natural selection. The basic idea is to make a population of candidate "programs" evolve toward the solution of a specific problem. Each individual of the population is evaluated by a fitness function that determines its ability to solve the target problem. New individuals are derived from existing one by applying genetic operators such as reproduction and crossover.

For our specific problem, we introduce the GenLTL algorithm to progressively learn a set of temporal usage patterns in the form of LTL expressions. GenLTL explores the space of possible LTL expressions representing combinations of method calls on an API with logical and temporal operators. The exploration is guided by the potential of each candidate pattern to capture correctly repetitive complex sequences of library method calls from client programs. To determine this potential, LTL expressions are assessed using a model checker on the execution traces.

4.1. Approach Overview

Figure 2 provides a high-level overview of the evolutionary approach proposed in this paper. The first step of our approach is to generate the initial set of LTL patterns to start the evolutionary process. Then, the first set of patterns is evaluated on the available execution traces. For every pattern, we measure its frequency and a form of accuracy called confidence. These two measures allow us to rank the patterns. Following that, pairs of patterns in the current iteration are mixed and mutated using genetic operators to drive new patterns, which are added to the next iteration. This process is repeated for a fixed number of iterations.

In the remainder of this section, we give the details about the different component of our algorithm: (i) encoding and generation of LTL patterns, (ii), evaluation of of the patterns, and (iii), the derivation of new patterns from the current ones during the evolution.

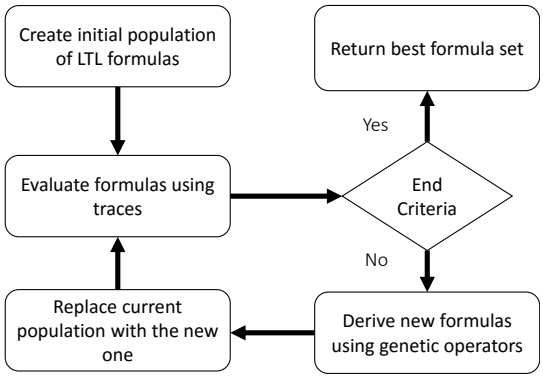

Figure 2: Approach overview

4.2. Encoding and generating LTL Patterns

LTL patterns are represented as an unbalanced binary tree. The tree contains two types of nodes: Operators (inner nodes) and Variable (leaf nodes). The tree is built so that the subtree(s) of a given node represents that operator's operand(s), as shown in the example of Figure 3.

- An operator node represents one operator accepted in LTL syntax and has one (unary) or two (binary) child nodes. The internal representation of the node's operator contains its name, symbol and type (Unary or Binary).
- A variable node refers to a method in the traces, which is represented by a symbol for legibility and performance reasons.

The tree is built such that an in-order traversal of the tree gives the syntactically correct representation. For that purpose, children of unary operator nodes are considered right-children with no left-sibling.

Figure 3 shows an example of a tree representing a pattern. The LTL pattern represented is

$$G(m1 \rightarrow XFm2) \wedge G(\neg m2 \cup m1)$$

This pattern means a call to the API method m1() is always followed by a call to the API method m2(), and method m2() cannot be called before the method m1()

The initial population is composed of N temporal patterns. This initial set of LTL patterns is generated following a top-down recursive algorithm to build patterns'

Table 1: Blocks used in the pattern generation algorithm

LTL Pattern	Logical Meaning
$G(x \rightarrow Xy)$	Event x always directly followed by event y
$G(x \rightarrow X!y)$	Event x never directly followed by event y
$G(x \rightarrow XFy)$	Event x always eventually followed by event y
$G(x \rightarrow XG(!y))$	Event x never eventually followed by event y

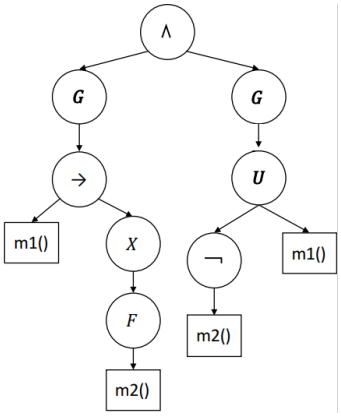

Figure 3: LTL binary tree

trees. An important aspect of the pattern generation algorithm is the use of "basic blocks", which are small, two-variable LTL expressions that are inserted at the bottom of generated pattern trees. These blocks represent linear correlations of events that have a meaning in the context of stack traces. They are included in the generated trees as to ensure that the initial LTL is at least somewhat sensible in the context of execution traces. Blocks are randomly selected from a small set (see Table 1). It is important to note that all blocks only contain free variables when inserted. Once the tree is built, all variables of the formula are randomly bound to the method calls in the traces. We ensure, however, that two instances of the same variable must share the same binding. We additionally check two integrity conditions: *block intersection* and *block triviality*.

- The block intersection condition specifies that each block of a single pattern must share, at least, one variable with one of the other blocks of the pattern. For example, the following pattern $(a \rightarrow Xb) \vee (b \rightarrow XF(c)) \wedge (d \rightarrow XG(\neg a))$ respects the condition, whereas that one does not respects the condition since the third block is completely independent from the rest of the pattern $(a \rightarrow Xb) \vee (b \rightarrow XF(c)) \wedge (d \rightarrow XG(\neg e))$
- The block triviality condition specifies that no block of the pattern may have twice the same variable. As such, blocks like $(a \rightarrow XFa)$ are rejected since they are usually either trivially true or trivially false.

4.3. Evaluating LTL Patterns

The evaluation of a potential LTL pattern consists in assessing its ability to capture the regularities in traces used in the mining process. This assessment is performed using Texada [15], a model checker. In this section, we starts by introducing Texada. Then, we detail the fitness function used in the evaluation.

4.3.1. Texada

Texada is a model checker that dynamically mines temporal specification in LTL from events or log. It can be used in different ways. The way we used in this work is to take an LTL pattern, i.e., an LTL formula, and check for all its occurrences in the traces. This process is done by evaluating three metrics: support, potential support and confidence. These are defined as follow:

- Potential support: For a pattern p, the potential support is the number of events in the traces that could falsify p. For a pattern *a call to method* $m1$ *is always followed by a call to method* $m2$, the potential support counts the number of times $m1$ is called in the traces.
- Support: support is the number of events that could falsify p, but do not falsify p. For our example, the support is the number of times $m2$ is called right after $m1$.
- Confidence: For a pattern p, confidence is the ratio of support over potential support.

To illustrate the calculation of the three metrics, let us consider the LTL pattern $f : G(a \rightarrow XG(b \rightarrow XFc))$, or with a short representation $(a, b) \rightarrow c$. This pattern means that c should happen after every sequence of a, b (even if other events occur between a and b). Consider now the trace set T composed by five traces t_0, t_1, t_2, t_3, t_4 shown in Table 2.

Table 2: Trace set T

Trace	events
t_0	a,b,c
t_1	a,d,b,c
t_2	a,b,d,c
t_3	a,d,b
t_4	d,b,a,c

Table 3 summarizes the calculation of the potential support (events underlined) and support (event in boldface following the underlined events) for the trace set T. This results in a potential support of 4, a support of 3 and a confidence of $3/4 = 0.75$.

4.3.2. Fitness Function

To evaluate the fitness of LTL patterns, we have to evaluate the "performance" of each pattern on the execution

Table 3: result summary

Trace	events	Potential support	Support
t_0	a,b,**c**	1	1
t_1	a,d,b,**c**	1	1
t_2	a,b,d,**c**	1	1
t_3	a,d,b	1	0
t_4	d,b,a,c	0	0
T		4	3

traces.

The definition of the fitness function is inspired by the two following heuristics:

- Penalize patterns that under-support. Patterns with low support should be penalized compared to other patterns. Since low support patterns, while they can have high confidence, are usually more specific patterns that are not easily generalizable. As such, they are less interesting than patterns with high support. In our experiments, we consider that a pattern under-supports if the potential support is less than 3.
- Penalize patterns that over-support. Patterns with extremely high support are trivial. In our experiments, we consider that a pattern over-supports if the potential support is greater than the trace set length.

For these reasons, the fitness function should separate the patterns in three "layers". The bottom layer contains patterns that over-support, the middle layer for patterns that under-support and the top layer for those with a good-support. These layers are created by a static fitness boost given to each category. The boost for the over-support and under-support categories is 0, and for good-support category is 0.5.

We therefore define the fitness function of GenLTL for a pattern p and a trace set T as:

$$fitness(p) = boost(p) + 0.5 * confidence(p)$$

where *boost* is defined as:

$$boost(p) = \begin{cases} 0 & if \quad support\_potential(p) < 3 \\ 0 & if \quad support\_potential(p) > length(T) \\ 0.5 & if \quad support\_potential(p) \in [3, length(T)] \end{cases}$$

4.4. Genetic Operators

To create the next generation, the algorithm relies on genetic operators to generate new patterns from existing ones. In order to maximize the variability in the new generated patterns, the mutations are not applied uniformly at each generation. To generate a new pattern, two starting patterns are picked at random using one of two methods: *Fitness Proportionate* or *Tournament*.

New patterns are created by probabilistically applying genetic operators in two steps. The first step concerns the crossover with both seed patterns crossed over with probability c. The resulting patterns (no matter if crossed over or not) are then randomly transformed by one of the mutations with probability m.

Modified patterns must respect all rules of property-checking. Table 4 shows some examples of possible mutations with integrity checking. In order to conserve integrity across mutations, the modifications applied on existing patterns cannot alter the structure of the basic blocks within it. Blocks can be swapped and their bindings changed, but the structure must stay the same. To implement this restriction, all trees have a "breakpoint" node separating the blocks from the rest of the tree. With the exception of binding mutations, no operation may modify nodes under the breakpoint.

Crossover. As illustrated in Figure 4, the crossover operator takes two patterns as input and produce two new patterns as output. The principle of this crossover is to select a subtree in each of the two patterns and switch them from one pattern to the other. Subtree of node α in tree A becomes subtree of node β in tree B and subtree of node β in tree B becomes subtree of node α in tree A. It is important to note that if a root node is picked as a switch node, the whole tree will be switched. When two subtrees are chosen to be swapped, the algorithm verifies that the cutting point is not located inside a basic block (or under a breakpoint as it may). Only when two valid cutting points are found are the subtrees swapped. Once the swapping is complete, both trees' integrity are checked. If either tree break block triviality or block intersection, the bindings are reworked (see binding Mutation below).

Operator Mutation. The operator mutation simply picks a random operator from the tree and replaces it by another operator of the same type. Unary operators are replaced by unary operators and binary operators by binary operators. However, in order to preserve block integrity, this mutation cannot modify operators inside basic blocks (effectively under the breakpoint).

Encapsulation Mutation. The encapsulation mutation simply takes an LTL tree and adds a unary operator at the top, that operator becoming the new root of the tree. This effectively takes an LTL pattern and surrounds it by a unary operator. For example, the pattern $x \rightarrow Xy$ could be mutated into $G(x \leftarrow Xy)$ should the G operator be applied to it. The encapsulated subtree may not be inside a basic block (A basic block can, however, be encapsulated with the new operator over the breakpoint).

Binding Mutation. The binding mutation simply changes the binding event of a variable for another event. Once the binding is changed, both block triviality and block intersection are checked. If block triviality is broken, another binding is chosen. If block intersection is broken, a connection is renewed using the new bindings.

5. Evaluation

5.1. Research Questions

To evaluate the efficiency and relevance of our approach, we defined four research questions:

Table 4: Example of possible mutations with integrity checking

Mutations	Original pattern	New pattern
Operator	$G(x \rightarrow Xy) \vee G(x \rightarrow Xz)$	$G(x \rightarrow Xy) \wedge G(x \rightarrow Xz)$
Encapsulation	$G(x \rightarrow Xy) \vee G(x \rightarrow Xz)$	$G(G(x \rightarrow Xy) \vee G(x \rightarrow Xz))$
Binding	$(\text{“}eventA\text{”} \rightarrow X\text{“}eventB\text{”}) \wedge (\text{“}eventA\text{”} \rightarrow X\text{“}eventC\text{”})$	$(\text{“}eventD\text{”} \rightarrow X\text{“}eventB\text{”}) \wedge (\text{“}eventD\text{”} \rightarrow X\text{“}eventC\text{”})$
Crossover	$G(x \rightarrow Xy) \vee G(x \rightarrow Xz), G(a \rightarrow XF(z)) \vee G(z \rightarrow Xw)$	$G(a \rightarrow XF(z)) \vee G(x \rightarrow Xz), G(x \rightarrow Xy) \vee G(y \rightarrow Xw)$

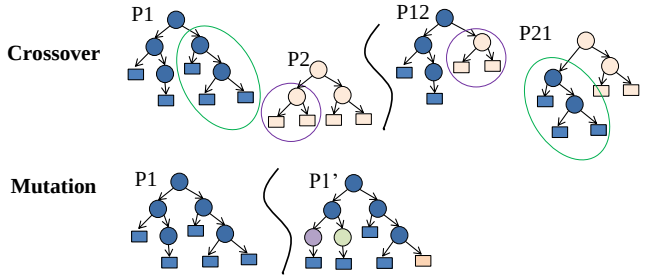

Figure 4: Genetic Operators

- **RQ1:** What kind of patterns we can infer with our approach?
- **RQ2:** Are the inferred patterns generalizable to other “new” client programs that are non-seen in the mining process?
- **RQ3:** Are the inferred patterns meaningful for developers?
- **RQ4:** What kind of LTL patterns are mined with a non-evolutionary state-of-the-art approach?

5.2. Experimental Setup

5.2.1. Data

We evaluate our technique through the usage of 8 widely used APIs from the Android platform: database, graphics, hardware, text, util, view, webkit, widget.

To perform our study, we selected 45 mobile apps using these APIs as shown in Table 5. To select the mobile apps, we opted for diversifying the validation dataset as much as possible. Thus we selected Android apps belonging to different domains. To assess the generalizability of the patterns we selected Android apps using multiple APIs. Table 6 show the distribution of the selected apps across the different APIs as well as Android categories. The APIs usage was inferred from the execution trace of typical usage scenarios for each app. Traces are an input of our methods and they should follow the structure required by Texada, which is a global liner input trace composed of multiple traces, each of which is composed of a totally ordered sequence of events (API method call) and traces should be separated with a trace separator. We provide the used traces for replication purposes and for more information on the amount of data collected for each API [1]. To avoid experimenter biases, we used independent usage scenarios defined in a previous study on energy consumption of API methods [16]. The traces are derived from 45 clients using the considered APIs. The original traces were processed to generate API specific traces based on the entry and exit time of each API method. As we are interested only in calls client-to-API, we only kept the first nesting level, which means that if an API method m2 is called from an API method m1, we do not consider it in the derived trace.

5.2.2. Procedure

For each research question, we present the research procedure to answer it. To address **research question RQ1** we run GenLTL on the whole dataset of traces. The termination condition of the learning process is when we reach the maximum number of fitness evaluations, which is 15000, over a horizon of 50 generations with a population size of 300. Then we only consider patterns with the perfect fitness score at last iteration (i.e. 1.0). There are no general rules to determine these parameters, and thus, we set the combination of parameter values by trial and error method, which is commonly used in the SBSE community. We opted for a large population rather than many Generations and our intuition was that the probability that these individuals produce interesting offspring increases with the population size thus larger populations can allow better crossover combinations.

To evaluate the structure of the inferred patterns and how complex they are, we compute the following metrics presented *TODO: Bad line break after table. FIXED* in Table 7. Number of Events is the number of method calls found in the trace data for the given API. Then we show the number of traces collected from client apps for each API. The coverage present on average the percentage of methods in the trace covered by the patterns methods. Number of patterns is the number of distinct LTL expressions mined in the trace of a given API. Methods per pattern is the average number of distinct methods per pattern. Pattern Depth is in average, the highest number of nodes from the root of the LTL tree to the beginning of a basic block. Pattern width is the average number of leaves in the patterns' tree. The width says on average how many methods are used in the patterns since potentially the same method may be used more than one time. Finally Average support as defined in the previous section.

The mining of usage patterns for an API depends on the used set of API's client programs (training client pro-

[1] http://geodes.iro.umontreal.ca/publication_material/ist19/

Table 5: Dataset used in the experiment

Apps	version	widget	webkit	view	util	text	hardware	graphics	database
Textgram	2.3.15			X		X			X
Antivirus Free	6.11.6			X	X	X			X
Simple Weather	1.1.3			X	X	X			X
Opera Mini web browser	7.5.3							X	X
News swipe	1.0.0	X		X	X	X			X
Easy Birthday Reminders	1.2.1			X	X	X		X	X
8.500+ Drink Recipes	1.0.6			X	X				X
Anti Mosquito Sonic Repellent	–		X	X	X	X			X
Arcane legends	1.0.7			X		X			X
Write Now Notepad	1.1.5			X				X	
Better Browser	2.3		X	X				X	
gReminders	0.9.7			X				X	
Dr.Web Antivirus Light	–	X		X				X	
10.000 Quotes DB (FREE!)	3.0.4			X	X			X	
Bubble blast 2	1.0.3			X				X	
Livo Recorder Lite	3.7.0			X	X			X	
MasterCard ATM Hunter	1.4	X		X				X	
SimpleNews	1.4			X				X	
Activity Express Task Manager	1.22			X				X	
5001 Amazing Facts Free	3.2.0		X	X	X			X	
AnEq Equalizer Free	1.0.9		X	X				X	
Wifi Radar	1.06	X	X		X	X	X	X	
Botanica	1			X				X	
Advanced Task Manager	2.1.2		X	X	X			X	
Anti dog mosquito whistle	1.3			X	X			X	
Better Notepad	0.0.5			X					
Map quest	1.8.1				X		X		
Inspiring Quotes	1.2				X		X		
Despicable me (minion rush)	1.1.0					X			
Droid Wifi Analyzer	1.3				X				
Battery HD	1.16			X					
Icey Slot	2.9			X					
Oxford AZ of English Usage	4.3.0			X					
aTimer	1.3			X					
Classical Music Radio Lite	1.0.3			X					
TED	2.0.1			X					
Droid Notepad	1.11			X					
Rome	–			X					
AndRecorder Free	3		X						
Sleep Sound Aid	–		X						
AudioPlayer	1.2		X	X					
25000 Best Quotes	1.0.7		X						
fAnime Radio Online	1.06		X	X					
Meridian Media Player Revolute	2.4.5		X	X					
World Travel Guide by Triposo	2.1	X							

Table 6: Distribution of 45 apps across APIs and categories

API	Apps	Category	Apps	Category	Apps
widget	5 (11.11%)	ARCADE	1 (2.22%)	FINANCE	1 (2.22%)
webkit	12 (26.67%)	BOOKS_AND_REFERENCE	4 (8.89%)	HEALTH_AND_FITNESS	2 (4.44%)
view	35 (77.78%)	BRAIN	1 (2.22%)	LIFESTYLE	1 (2.22%)
util	15 (33.33%)	BUSINESS	1 (2.22%)	MEDIA_AND_VIDEO	2 (4.44%)
text	9 (20.00%)	CARDS	1 (2.22%)	MUSIC_AND_AUDIO	5 (11.11%)
hardware	3 (6.67%)	CASUAL	1 (2.22%)	NEWS_AND_MAGAZINES	3 (6.67%)
graphics	18 (40.00%)	COMMUNICATION	2 (4.44%)	PRODUCTIVITY	4 (8.89%)
database	9 (20.00%)	EDUCATION	1 (2.22%)	TOOLS	10 (22.22%)
–	–	ENTERTAINMENT	2 (4.44%)	TRAVEL_AND_LOCAL	3 (6.67%)

grams). Hence, to address our **research question RQ2**, we need to evaluate whether the detected API's usage patterns will remain with similar confidence degree in the context of new client programs of the API (validation client programs). Our hypothesis is that: detected usage patterns for an API are said "generalizable" if they remain characterized by a high confidence degree in the contexts of various API client programs. This is regardless of the natures and features of those client programs, and of whether those programs were used or not for detecting the API's usage patterns.

To evaluate the generalizability of detected patterns, we perform leave-one-out cross-validations for all the selected APIs while considering the client programs using each API in the considered set of 45 mobile apps selected for our study. Let N represents the number of used client programs for the considered API (e.g., N = 37 for the view API), we perform N runs of GenLTL on the API. Each run uses N-1 client programs as training client programs for detecting usage patterns and leaves away one of the APIs client programs as validation client programs. The results are sorted in N runs, where each run has its associated usage patterns and its corresponding training and validation client programs. We use the leave-one-out strategy instead of tradition k-fold, because it represents realistic scenarios for our problem. Indeed, the usage scenario of our approach corresponds to a situation where patterns learned from a reasonable set of existing API clients should apply to a new client under development. Additionally, leave-one-out is suitable when the set of training data is relatively small, which is our case.

Then, we address our second question (RQ2) in two steps, as follows. In the first step, for each run of the cross-validation, we selected the inferred patterns that have a confidence degree upper than a certain threshold (in our experiment the threshold was 80%). In the second step, for each selected pattern we use the Texada tool to check if the pattern actually holds in the trace of the validation client and with which confidence degree. The goal is to see if good patterns remain good in the context of new client programs.

To answer **research question RQ3**, *i.e.*, evaluate the meaningfulness of the mined patterns from a human perspective, two authors analyzed qualitatively the inferred patterns generated for a database API (`android.database.sqlite`). We focused on a single API so that evaluators could familiarize themselves to a reasonable degree with rich, in-depth information taken from its documentation.

The goal behind **research question RQ4** is to compare our approach with a baseline approach, through shedding light on the kind of patterns that it could infer.

We selected the approach proposed by Lo *et al.* [18] as it presents a different and very interesting idea for mining non-trivial API usage patterns. Moreover, to the best of our knowledge, the select approach is most similar to our approach from the perspective of the required input and the produced patterns. The selected approach, mines a set of temporal rules of arbitrary lengths from program execution traces. Then it represents the mined rules as LTL expressions, so that existing model checkers could consume the mined rule [18] .

We implemented the approach proposed by Lo *et al.*, as best as we could according to the description and details provided in [18]. This implementation aims at mining patterns in the form of:

$$(a, b) \rightarrow (c, d)$$

which reads, whenever a series of pre-conditions events (a,b) occur, eventually, another series of post-condition events (c,d) will occur. The set of pre-conditions and post-conditions events satisfying a minimal support and confidence thresholds are mined using BIDE [36], the closed sequential pattern miner.

The original approach used a miner modified from BIDE, and we were not able to have access to the details of the modification. Moreover, the original approach defined a filtering step to discard rules that could be generalized to other rules and keep only a minimal subset of rules. This filtering step added a significant over had to the mining process and thus we decided to avoid it. These two differences make the resulting implementation, more an inspiration of the original approach rather than an exact implementation.

5.2.3. Results

RQ1: *What kind of patterns we can infer with our approach?*

A cursory look at the results of Table 7 reveals some interesting details that merit an in-depth analysis. First, it is worth to mention that we show the characteristics of solutions found (*i.e.* patterns with fitness 1.0 at last iteration). The inferred patterns for all studied API have a high average confidence that reach 100%. These high confidences seem to lend credence to the idea that most patterns found with GenLTL will be reliable since, in most cases, they will be respected in the API properties. If, by opposition, the confidence were to be found low, this would cast doubts on the usability of found patterns since, even if interesting, the found patterns wouldn't be respected most of the time.

A second noteworthy detail is the consistently high amount of support for each API (last column in Table 7). This support reinforces the usability of patterns found through the GenLTL algorithm since it implies that the found pattern is usually used throughout the API and not at one single point in the API. Patterns with such support would then be trivial to generalize; something that is not easily done with low support patterns. For example, a pattern with a confidence of 100% but support of 1 would be of little interest since, even if it has a high confidence, the low support would indicate that the pattern is used sparingly through a typical use of the API and therefore, hard to generalize.

A third interesting detail found in the results is the fact that the graphics and util APIs have low coverage (8%) compared to other APIs (the fourth column in Table 7). The presence of such low coverage amidst the results seems to indicate that GenLTL can not only find "general" patterns (patterns concern a big portion of the API) but also "local" patterns (patterns that concern specific set of methods in API). A last detail of note is the fact that patterns found in the database API are, on average, more complex than those found in other APIs. Indeed, the number of methods per pattern, the depth and width of patterns (respectively the 6th, 7th and 8th column of Table 7) are all higher in this API as compared to others. This fact seems to indicate that a bigger number of methods are interdependent in this API and that their usage more closely correlate to one another. This can probably be explained by the fact that the database API have more methods that need to be called together for the whole API to function.

RQ2: *Are the inferred patterns generalizable to other "new" client programs that are non-seen in the mining process?*

In Figure 5 we plot for each API the confidence degree of the selected pattern in both the training or learning process API_L and the validation or testing process API_T.

The boxplots show that a very few degradation in the confidence degree can be observed between training and validation context. Actually, except for database, webKit and widget, the confidence degree was almost equal to 100 % for all the other APIs in both contexts which reflect a very high generalizability for the mined patterns. The worst degradation in the confidence degree was noticed for the widget API, this could be explained by the fact that the widget package contains mostly UI elements to use to create an application widget, and most of the apps can have different widgets and we can even have the same app proposing different widgets to its end user, with means the use of this API may vary from an application to another. However, despite we had the worst degradation for the widget API, the confidence degree remain over 90% in both training and validation context. Finally, hardware API shows 100% confidence degree. This is due to the reduced number and the triviality of its traces whose distincts methods are only 4 and come over-repeatedly in the traces. Nevertheless, this corner case figures how applicable is our approach on real world (often erratic) cases.

These results show that the mined patterns with our technique can be used to enhance the API documentation with high confidence without a need to consider all possible usage contexts (client programs) of the API of interest.

RQ3: *Are the inferred patterns meaningful for developers?*

At the start, the evaluators did a calibration session, where they analyzed 7 patterns together and established the following procedure:

a) Summarize the structure of the pattern and identify its components.
b) Search in the API documentation for a 1-line description of each method involved in the pattern.
c) Write a description of each component in natural language.
d) If the descriptions of each component are meaningful, write an overall natural language description (a "story") for the entire pattern.
e) Indicate whether the story is "sensible" or "not sensible".

Using this procedure, each evaluator examined independently a sample of 22 additional patterns with support value between 5 and 823. The first evaluator classified 12 out of 22 patterns as sensible; the second evaluator 15 out of 22. There were 3 instances where the two evaluators expressed conflicting opinions, and the Cohen's Kappa coefficient was calculated as 0.53, indicating moderate inter-evaluator agreement. These observations are preliminary evidence that GenLTL can produce meaningful patterns. To further illustrate this, we describe some characteristic cases.

First consider the pattern P_1:

$$G(c \rightarrow XG(\neg b)) \oplus G(c \rightarrow X\neg a)$$

The variables in P_1 are the methods of classes in the API, listed in Table 8. P_1 is an exclusive disjunction (XOR) decomposition of two base block subformulas. The first subformula describes the constraint that an update operation

Table 7: Descriptive statistics of our setting and results

	Traces		Patterns					
API	Nb events	Nb traces	coverage	Nb patterns	Nb method PerPattern	depthPattern	widthPattern	support
database	4057	10	0.44	124	6.07	2.40	7.00	567.47
graphics	1210	21	0.08	115	5.26	2.23	6.34	278.05
hardware	559	3	0.47	80	3.33	2.42	6.76	158.13
text	483	12	0.21	89	3.87	2.37	6.99	101.28
util	878	27	0.08	115	5.36	2.34	6.74	168.64
view	1255	45	0.12	115	5.15	2.24	6.59	335.73
webkit	320	15	0.33	110	4.78	2.51	7.09	28.47
widget	444	5	0.31	94	3.99	2.23	6.45	86.40

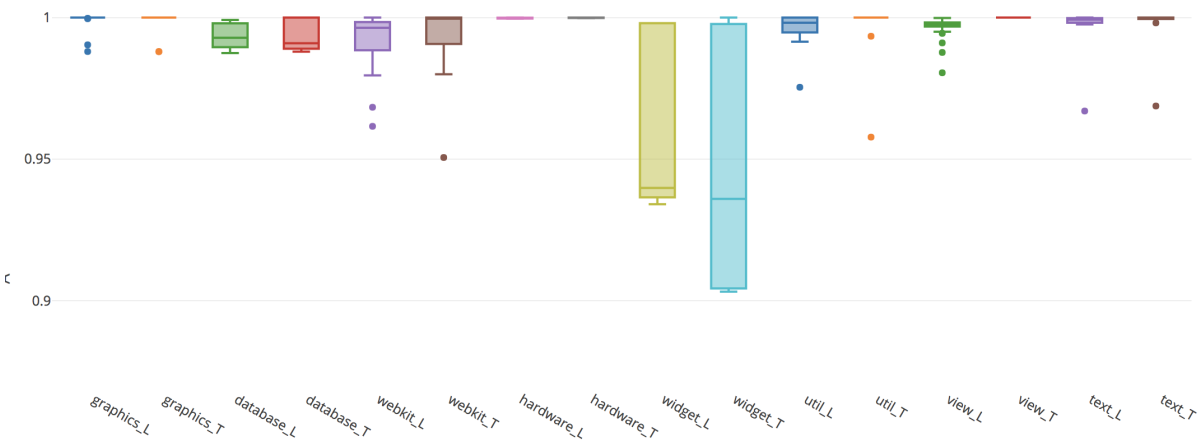

Figure 5: Patterns Generalizability

Table 8: Variables used in qualitatively analyzed mined patterns.

Variable	Methods in P_1
a	`SQLiteClosable.close()`
b	`SQLiteOpenHelper.onOpen()`
c	`SQLiteDatabase.update()`
Variable	**Methods in P_2**
p86	`SQLiteQueryBuilder.query3()`
p48	`SQLiteDatabase.compileStatement()`
p67	`SQLiteOpenHelper.SQLiteOpenHelper()`
p85	`SQLiteQueryBuilder.query1()`
p48	`SQLiteDatabase.compileStatement()`

Parameter types omitted for brevity. The overloaded methods query3 and query1 are indexed by the order by which they appear in the API documentation.

should never be followed by a state where the database is being opened. The second subformula describes the constraint that an update operation should never be immediately followed by a close operation. Since the XOR composition of those two blocks implies that only one of the two block must be true at any time time, the resulting pattern "story" can be expressed as follows: After calling `update`, if we call `close`, then `onOpen` will never be called. This is a sensible scenario as one would not call a `onOpen` method after closing a database. In the opposite case, if `close` is not called right away, it makes sense that `onOpen` would eventually be called.

We also consider the more complex pattern P_2:

$$(G((p86 \rightarrow Xp48) \rightarrow XXp67) \rightarrow ((p85 \rightarrow \neg XFp85)UXp48))$$

The variables in P_2 are the methods of classes in the API, listed in Table 8. P_2 consists of two component subformulas, connected by an implication operator. The first subformula describes a sequence of operations: the execution of a query, followed by the compilation of a statement, followed by the creation of an helper object, used for the opening of new database instances. The second subformula defines a constraint: the execution of a query cannot be repeated until a new statement is compiled. The implication between the two subformulas thus expresses the following "story": If the method used for extracting data from the database involves compiling a new statement and opening a new database object after every query, then compiled queries cannot be reused.

These two examples illustrate vividly the kinds of rich behavioural patterns that can be created with GenLTL.

RQ4: *What kind of LTL patterns are mined with a non-evolutionary state-of-the-art approach?*

We run the implemented baseline approach on the dataset of traces considered for RQ1, we started with min-

imal support and confidence thresholds of 80%. For each API we either obtain the results within an hour, or we relax the initial thresholds with a step of 10% and we restart the mining process. At the end, we were only able to infer patterns for the APIs view, webkit, graphics, and util. Results are presented in Table 9, to have an idea of the complexity of inferred patterns we present the following metrics. The number of distinct patterns mined in the trace of a given API, the pattern depth and width as defined for RQ1. as well as the first values of the input parameters minimal support and confidence for which we could infer patterns from the considered API Trace.

For high values of support and confidence parameters, the inferred patterns are supposed to be generalizable to other client systems, However, we noticed that for such values the inferred patterns are rare. Moreover, although the technique is meant for mining nontrivial patterns, we notice that the inferred patterns are on average formed of 3 methods and reached a maximum depth of 2 nodes. the inferred rules s are of course more complex than the standard two-event rules but are still less complex than the ones mined with our technique GenLTL. In addition from expressiveness perspective, the mined rules are limited to the following expressions whereas in the case of GenLTL we are not constrained with a pre-defined reference of how the pattern might look like:

$$a \rightarrow b : G(a \rightarrow XFb)$$

$$(a, b) \rightarrow c : G(a \rightarrow XG(b \rightarrow XFc))$$

$$a \rightarrow (b, c) : G(a \rightarrow XF(b \wedge XFc))$$

$$(a, b) \rightarrow (c, d) : G(a \rightarrow XG(b \rightarrow XF(c \wedge XFd)))$$

We also have to mention that most of the time it is hard to decide on the best parameters to mine each trace. For instance, as soon as we relax the minimal support and confidence thresholds the pre-conditions and post-conditions events of the mined rules could be of arbitrary lengths which could drastically increase the complexity of the patterns. For example, with 70/70 as minSupport/ minConfidence input, we mined a rule for the view API with 11 methods as a width of the pattern, and we were unable to check it with Texada.

Threats to Validity. Even though we performed experiments on 8 different APIs, the choice of APIs may pose threats to the generalizability of our conclusions, as they all belong to the Android platform. In future work, we plan to evaluate our approach on APIs and client systems belonging to different domains, having different sizes and coming from different organizations.

An additional threat to validity is posed by the assumptions underlying our choices of tools and mechanisms used in our experiments and technique. We completely rely on the Texada tool [15] to compute important metrics such us the pattern support and confidence. This could impact the extent to which our actual observations corresponded to the phenomena that we intended to observe, this posing a threat to construct validity. To mitigate this threat, three of the authors tested the tool separately on different basic and complex examples to check that it actually performs what is mentioned in its documentation.

Experimenter expectancy effect is another possible threat to validity. Indeed, for RQ3, the manual inspection was performed by two of the authors. To attenuate this threat, we followed a rigorous process with a calibration session. To have a more objective evaluation, we plan to replicate the study with independent subjects.

It is to mention that the algorithm takes between 3 to 45 minutes to perform. Yet, mining patterns is not meant to be run in a real-time context and therefore remains practicable in a real world application.

Another potential threat to the validity of our results is related to the relatively high number of solutions explored by our algorithm when searching for patterns. To assess whether our results are attributable to our approach or to the number of solutions explored, we performed a sanity check using the database API. We compared GenLTL to random search on three independent runs. In each run, we generated randomly a number of solutions equal to the number of solutions explored by GenLTL (population size * number of generations). The random solutions contain patterns with equivalent confidence values than those of GenLTL. However, these patterns are very simple, i.e., temporal relations between two methods, whereas, the patterns of GenLTL capture more sophisticated temporal relations involving many methods. We are, then, confident that random search cannot generate complex patterns with the same level of confidence than GenLTL.

6. Usage of API Temporal Patterns

We have created the tool Tapir[2] (*Temporal API Recommender*) that guides developers in using APIs based on mined temporal patterns. In this section, we present how it integrates temporal patterns into development activities. Specifically, Tapir focuses on four development activities where we can take advantage of the LTL patterns as illustrated in Figure 6:

- *a)* During testing, Tapir can verify whether execution traces of API client code satisfy the mined patterns.
- *b)* At compilation time, it can assess whether API client code satisfies the pattern using static analysis to warn users of potential problems.
- *c)* During programming, it can provide developers with warnings, as well as recommendations for completing client code with API calls.
- *d)* Finally, we envision using Tapir to augment existing API documentation by interpreting the automatically mined patterns into structured natural language.

[2]https://bitbucket.org/ErickFifa/tapir/

Table 9: Descriptive statistics for the implementation of Lo *et al.* [18]

API	NB patterns	depthPattern	widthPattern	minSupport/ minConfidence
view	1	1	2	80/80
graphics	2	1.5	2.5	60/60
util	9	1.8	2.8	30/30
webkit	2	2	2	80/80

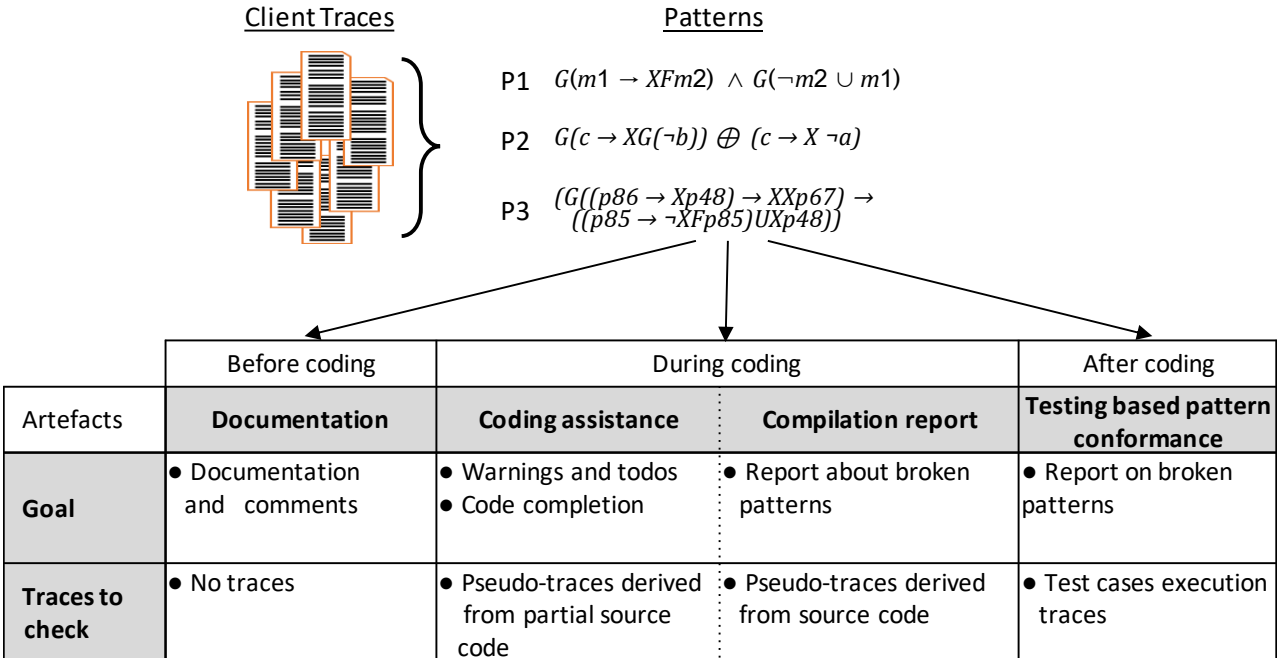

Figure 6: Tapir use cases.

In the following, we use a running example to illustrate how Tapir leverages mined API usage patterns in these contexts.

6.1. Running Example

We use the example of the CIDR Calculator Android app[3], that uses the `android.database.sqlite` API[4]. Specifically, we consider the class `HistoryProvider`. This class is responsible of creating, opening and upgrading a history database. Within the class, there are several `sqlite` API calls, specifically from the classes referred in Table 10.

We further consider the pattern P_{RE}, that was mined during the evaluation presented in Section 5:

$$(((p41 \rightarrow \neg XFp41)\ U\ Xp69)$$
$$\rightarrow$$
$$((p38 \rightarrow \neg XFp38)\ U\ Xp30))$$

Table 10: Variables used in running example pattern P_{RE}.

Classes	
`android.database.Cursor`	
`android.content.SharedPreferences`	
`android.database.sqlite.SQLiteDatabase`	
Variable	**Methods in P_2**
p41	`Cursor#close()`
p69	`SQLiteDatabase#getWritableDB(String)`
p38	`SharedPreferences#getString(int, String)`
p30	`Cursor#moveToPosition(int)`

The variables p41, p69, p38, and p30 are shorthand for API calls as shown in Table 10.

Effectively, P_{RE} is made up of two sub-patterns. The sub-pattern P_{RE1} defines a rule that a Cursor should not be closed twice before the method getWritableDB is invoked. The sub-pattern P_{RE2}, defines a rule that getString cannot be invoked twice before moveToPosition is invoked. Overall then P_{RE} says that if the rule P_{RE1} holds in a trace, so should the rule P_{RE2}. Effectively, if a file is reasonably managed by not being closed twice, then it is expected that reasonable usage also means not doing repeated reads from the same cursor position.

[3] `https://github.com/rmceoin/cidrcalculator`
[4] `https://developer.android.com/reference/android/database/sqlite/package-summary`

Listing 1: The method keepCheck().

```
private void keepCheck(SQLiteDatabase db) {
    if (debug) Log.d(TAG, "keepCheck()");
    long count;
    try {
        SQLiteStatement r = db.compileStatement("
        SELECT␣count(*)␣FROM␣" +
                HISTORY_TABLE_NAME);
                // Call of getWritableDatabase: p69
        count = r.simpleQueryForLong();
    } catch (SQLiteException e) {
        e.printStackTrace();
        return;
    }

    Context context = getContext();
    if (context == null) {
        return;
    }
    SharedPreferences sp = PreferenceManager.
    getDefaultSharedPreferences(context);

    String historyEntries = sp.getString(Preferences
    .PREFERENCE_HISTORY_ENTRIES, "100");
    int maxRows = Integer.parseInt(historyEntries);
    if (count > maxRows) {
        if (debug) Log.d(TAG, "keepCheck:␣greater␣
        than␣" + maxRows);
        Cursor c;
        try {
            String[] projection = {History._ID};
            c = db.query(HISTORY_TABLE_NAME,
            projection, null, null, null, null,
            History.DEFAULT_SORT_ORDER);
            c.moveToPosition(maxRows);
            while (!c.isAfterLast()) {
                int rowId = c.getInt(0);
                if (debug) Log.d(TAG, "keepCheck:␣
                need␣to␣delete␣" + rowId);
                db.delete(HISTORY_TABLE_NAME,
                History._ID + "=" + rowId, null);
                c.moveToNext();
            }
            c.close();
        } catch (SQLiteException e) {
            e.printStackTrace();
        }
    }
}
```

Listing 1 shows a relevant snippet of the method `HistoryProvider#keepCheck()`. In the following, we will be using it and the pattern P_{RE} to show how mined API patterns can be leveraged during software development by Tapir.

6.2. Testing Time

Given an execution trace t of a client, Tapir can verify whether t satisfies some mined API pattern p. Such execution traces are collected during code testing and are preprocessed to only keep events corresponding to API calls. Tapir takes as inputs the trace t and the pattern p and outputs a pair of values (*support*, *potential_support*), as defined in Section 4. If the result is (0,0), the pattern p is not relevant to the trace t, as none of the methods in p are being present in t. If the two values are equal and non-zero, the pattern p is relevant to t, and t satisfies p. Developers can choose whether to be informed about patterns that are relevant and satisfied. In the third option (different, non-zero values), the pattern p is relevant to t (i.e., t uses methods involved in p), however t does not satisfy p. This could be an indication of potential API misuse. In such cases, Tapir outputs a warning to the user for further manual analysis.

In the `keepCheck()` example, the code could be represented as shown in Listing 2 by filtering and indexing the relevant information. From this representation, it infers the trace t_1 shown in Listing 3 and produces the output is *support* = 4 and *potential_support* = 5. This result indicates that the pattern P_{RE} is relevant for this code fragment but is not satisfied. Thus Tapir produces a warning to alert the user of a potential misuse of the API in the `keepCheck()` method.

Future updates to Tapir will put the emphasis on closing the loop with the user. Since currently execution traces are tested based on patterns mined from existing client code, there may exist legitimate usages of the API that are not captured (or even contradict) the mined patterns. If a developer chooses to ignore a testing time warning generated by Tapir, we may need to refine the relevant pattern, in a fashion similar to the CEGAR loop [5]. However, this should be done judiciously to avoid malicious attempts to corrupt usage patterns.

Further, Tapir should allow evolving its API usage patterns based on empirical evidence. We envision keeping track of warnings for patterns that are ignored by developers too often, allowing us to reevaluate the usefulness of each pattern. Tapir could then attribute a severity score based on empirical data that integrates developer reactions to analysis feedback, similar to the Google Tricorder approach [25].

6.3. Compile Time

At compilation time we aim to determine whether the client code respects the mined API patterns without needing to execute it. We envisage doing that by statically analyzing client code to generate "pseudo-traces". Given a

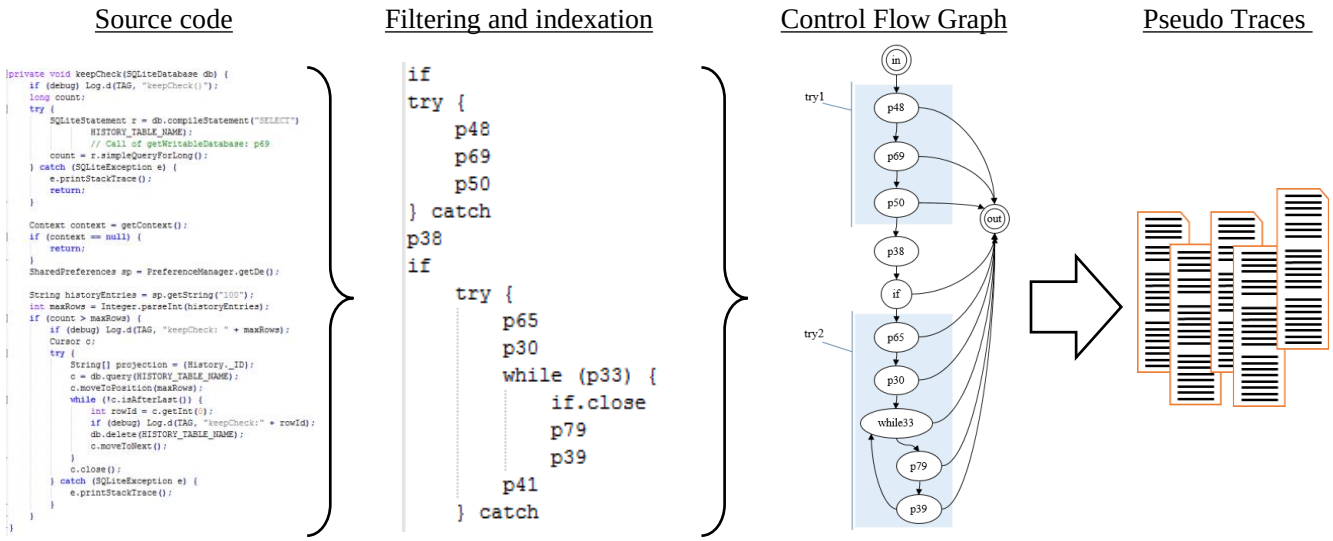

Figure 7: Pseudo-traces derivation from source code process

Listing 2: `keepCheck()` preprocessed: the method instructions are filtered and indexed to focus on control flow and SQLite API

```
if
try {
    p48
    p69
    p50
} catch
p38
if
    try {
        p65
        p30
        while (p33) {
            if
            p79
            p39
        p41
    } catch
```

Listing 3: t_V: a pseudo trace for method `keepCheck()` violating P_{RE} *i.e. support* = 4 and *potential_support* = 5

```
p48;p69;p50;p38;p65
```

Listing 4: t_{NR}: a pseudo trace for method `keepCheck()` non relevant to P_{RE} *i.e. support* = 0 and *potential_support* = 0

```
p48;p50;p38;p65;p30;p33;p79;p39;p41
```

Listing 5: t_S: a pseudo trace for method `keepCheck()` satisfying P_{RE} *i.e. support* = 7 and *potential_support* = 7

```
p48;p50;p38;p65;p30;p33;p79;p39;p41
```

pseudo-trace t and a pattern p, Tapir will use the approach outlined in Section 6.2, to generate appropriate warnings to the developer. These will be communicated along with other compile time messages.

To generate pseudo-traces, we plan to use the approach outlined in Figure 7. Specifically, Tapir will follow these steps:

a) filter the client code, keeping only control flow instructions (conditionals, switch statements, loops, try-catch blocks) and API calls. API calls are indexed to ease the process.

b) use the resulting code to construct a control flow graph (CFG) G.

c) compute the set of mutually independent paths on G. We call these pseudo-traces.

Implementation of this technique is ongoing work.

For example, in the case of the `keepCheck()` code shown in Listing 1, the corresponding CFG is shown in Figure 8. From this graph, the extracted pseudo-traces are t_V, t_{NR}, t_S, shown in respectively Listings 3, 4, and 5. The result of checking the pattern P_{RE} on t_{NR} is $(0, 0)$. This means that t_{NR} is not a relevant trace and no further action is needed. For t_S, the result is $(7, 7)$, indicating that t_S satisfies P_{RE}. A message can be generated if the developer has indicated that she wants to be informed about patterns that are used and satisfied. Finally, for t_V, the result is $(4, 5)$. Tapir will thus raise a warning to the developer indicating the existence of potential misuse of P_{RE}.

This approach is not immune to producing false positives and false negatives. Consider a case where some structural paths may be non-executed due to mutual exclusion control flow branches. Pseudo-traces will then contain unfeasible paths. We envision improving Tapir to reduce false positive and negatives by using block graph [13] and performing dead code detection [1].

Another limitation of Tapir is that it does not analyse

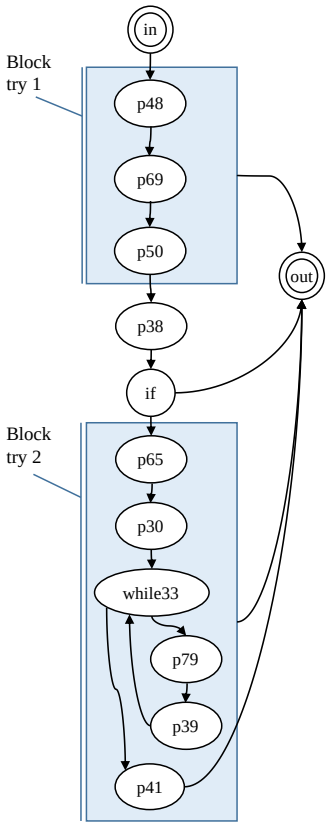

Figure 8: Control Flow Graph representing the method `keepCheck()`

interprocedural API usages. For instance, consider the case where Line 35 (`c.close()`) of Listing 1 (*i.e.* call `p41`) is replaced by a call to a helper method. The helper method contains the API call (p41), that, if inlined in `keepCheck()`, would result in P_{RE} being satisfied. However, the filtering step of Tapir removes it altogether, not recognizing that it constitutes indirect API usage. More generally, this problem of false negatives occurs when API usage is scattered across different client methods. This can be addressed by more sophisticated interprocedural static analysis [12].

6.4. Coding Time

Software developers routinely make use of functionalities provided by modern IDEs, such as code completion and incremental compilation. In the previous section, we described how Tapir could use CFGs to generate feedback for developers at compile time. However, during the time that a developer is in the process of coding, the code is by definition incomplete. During this stage, Tapir provides feedback to the developer in two contexts: (a) as IDE warnings generated during incremental compilation, and (b) as code completion recommendations.

Take for example the case where a developer wants to access a database for writing. It is required that she use the Android#SQLite API to create and/or open a database connection. This is done using the getWritableDatabase()

Listing 6: Example of an insert method

```
public static void insert(int ID, int val, int ref,
int id, int v1, int v2) {
ContentValues contentValues = new ContentValues();
contentValues.put(ID, id);
contentValues.put(ref, v1);
contentValues.put(val, v2);
SQLiteDatabase db = this.getReadableDatabase();
db.insert(TABLE_REF, null, contentValues);
db.close();
}
```

Table 11: Variables for F_c

Variable	Methods
rd	SQLiteOpenHelper#getReadableDatabase
in	SQLiteDatabase#insert
up	SQLiteDatabase#update

or getReadableDatabase() methods of the SQLiteOpenHelper class, as shown in Listing 6.

As stated in the API documentation[5] the SQLiteOpenHelper#getReadableDatabase() method might return an object database in read-only mode. In this case, invoking methods that write in a database, such as SQLiteDatabase#insert() or SQLiteDatabase#update(), will cause the program to not function properly. This problematic case can be expressed in the LTL formula $F_c : G(rd \rightarrow XF(in|up))$

Tapir uses Grapacc, a code completion Eclipse plugin [20], that produces code completion recommendations based on API usage patterns. Grapacc uses grouMiner [21] to build a database of frequent usages of APIS in form of *groum*s, a data structure that represents control and data flow in code fragments. Tapir generates a Grappac input file to start the Grappac plugin. The plugin in turn analyzes the code being edited, by internally encoding it as an incomplete groum and attempting to return the most similar pattern. Tapir uses the Grappac output to generate warnings of potentially problematic API usage. In case of multiple pattern violations, Tapir ranks them according to relevance, as shown in the illustrative example in Figure 9.

6.5. Documentation Time

Temporal API usage patterns can be used before the developer starts writing the client application code, by translating them into natural language documentation and comments. For example, in the case of P_{RE} we have already translated the LTL formula in a natural language description in Section 6.1. Developers planning to implement components that use the API calls implicated in P_{RE}, such as `keepCheck()` could benefit from such descriptions

[5] `https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html`

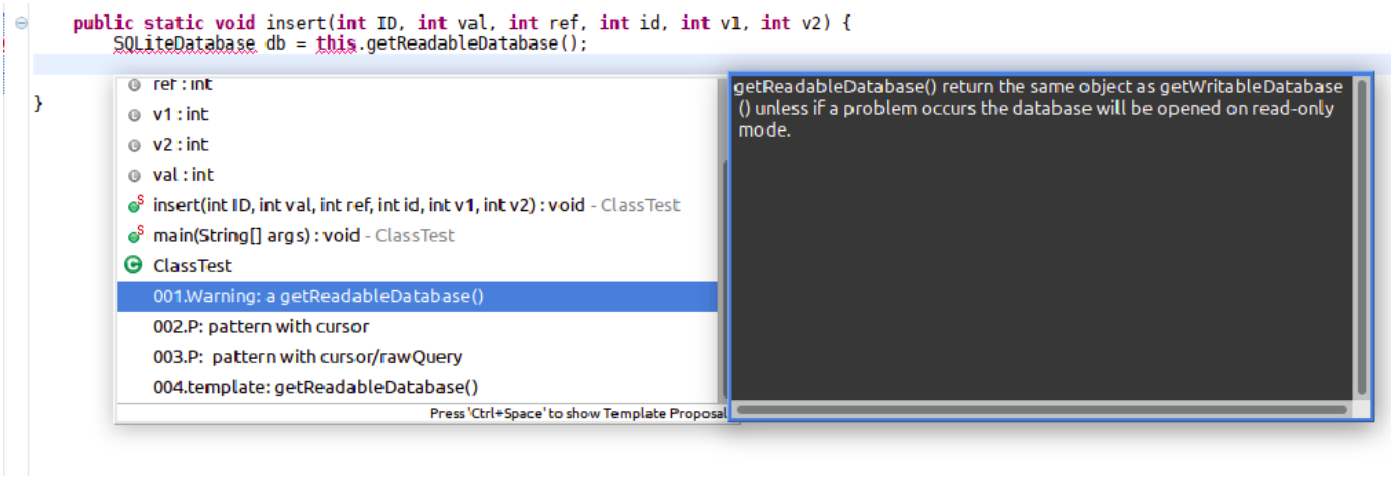

Figure 9: Tapir warning, integrated with code completion in Eclipse.

of the patterns in natural language. We envision using Tapir to generate such documentation.

In Section 2, we described the three classes of API usage patterns, classified by "usefulness": $\mathcal{H}$, containing patterns that can be understood by humans with reasonable cognitive effort; $\mathcal{M}$, containing patterns that, while not humanly comprehensible, can still be efficiently leveraged by automated techniques; and $\mathcal{I}$, containing patterns that cannot be practically used either by humans or automated techniques. Patterns in $\mathcal{I}$ are by definition outside the scope of our interest here.

Much work has been done in translating natural language specifications into LTL formulas (see, for example [19, 7]). However, to the best of our knowledge, little work addressed the translation of LTL formulas into natural language, beyond a set of well understood patterns [6]. Direct translation of LTL formulas may lead to verbose and unintelligible text. As a first step towards this approach, Tapir generates graphical representations of LTL formulas as Groums [21] to help developers understand the meaning of the patterns.

For the validation in Section 5, we manually translated for patterns, and the resulting text is in general more abstract and concise than the combination of individual meanings of the involved logical and temporal operators. The strategy we are working on to automate such a translation is based on the use of examples. First, we create a corpus of formulas, of different shapes and complexities, with their corresponding natural language texts. Then, this corpus is used in an example-based machine translation framework [32] to convert mined API temporal patterns into documentation.

Generating documentation for patterns in $\mathcal{M}$ is by definition much harder. A potential approach might be to try to generate only some characteristic traces and describe them along with their relationships. More imaginative solutions could include interactive training tools, that would allow developers to understand the patterns from direct experience.

7. Discussion and conclusion

We have proposed a genetic-programming approach to recover API temporal constraints from execution traces of client programs using the API. Our approach explores the space of LTL expressions, representing the candidate patterns, that can be defined on the API public methods. The exploration is guided by the applicability of candidate patterns to the trace samples. Unlike most of the existing approaches, ours does not search for specific pattern templates. We evaluated our approach on eight libraries. Our results show that we are able to recover a wide range of usage patterns in terms of size, complexity, and variety of temporal and logical operators. It also demonstrated that the recovered patterns are generalizable to clients not considered in the recovery process. Additionally, we assessed the meaningfulness of the recovered patterns. The majority of patterns in the analyzed sample were considered as meaningful with respect to the API functionalities. We believe our work is an important stepping stone towards the assistance of developers in safely using the multiple APIs necessary to their development tasks. Indeed, the recovered patterns, when they are humanly understandable, can be used to document the API. When these patterns are too complex, they can still be useful to automatically recommend usage scenarios or detect suspect and incorrect situations as outlined in the proposed agenda for integrating temporal patterns into development activities.

Although the obtained results are very encouraging, there is room for improvement. First, the approach can be improved to avoid producing candidate patterns that are too trivial, especially when the negation operator is used. Another possible improvement concerns the fitness functions. Currently, we use a single function that combines the support and the confidence with threshold values. Defining accurate values for these thresholds is

not obvious. An alternative option to explore is to use a multi-objective search. From the evaluation perspective, it is necessary to have a larger study on the readability and the usefulness of the recovered patterns. Such a study should involve actual developers using a set of APIs.

Acknowledgements

We acknowledge the contributions of Pierre-Olivier Talbot in the conference version of this paper. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Camila Bastos, Paulo Afonso Junior, and Heitor Costa. 2016. Detection techniques of dead code: Systematic literature review. In *Proceedings of the XII Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era-Volume 1*. Brazilian Computer Society, 34.

[2] Omar Benomar, Hani Abdeen, Houari Sahraoui, Pierre Poulin, and Mohamed Aymen Saied. 2015. Detection of software evolution phases based on development activities. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 15–24.

[3] Ervina Çergani, Sebastian Proksch, Sarah Nadi, and Mira Mezini. 2018. Investigating Order Information in API-Usage Patterns: A Benchmark and Empirical Study. In *Proceedings of the 13th International Conference on Software Technologies, ICSOFT 2018, Porto, Portugal, July 26-28, 2018.*, Leszek A. Maciaszek and Marten van Sinderen (Eds.). SciTePress, 91–102.

[4] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching Connected API Subgraph via Text Phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, Article 10, 11 pages.

[5] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–169.

[6] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 411–420. DOI:http://dx.doi.org/10.1145/302405.302672

[7] J. Dzifcak, M. Scheutz, C. Baral, and P. Schermerhorn. 2009. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *IEEE International Conference on Robotics and Automation*. 4163–4168.

[8] Mark Gabel and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. 339–349.

[9] Mark Gabel and Zhendong Su. 2010. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 15–24.

[10] Heather J. Goldsby and Betty H. C. Cheng. 2010. *Automatically Discovering Properties That Specify the Latent Behavior of UML Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 316–330. DOI: http://dx.doi.org/10.1007/978-3-642-16145-2_22

[11] Samuel Huppe, Mohamed Aymen Saied, and Houari Sahraoui. 2017. Mining complex temporal API usage patterns: an evolutionary approach. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 274–276.

[12] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2010. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*. Springer, 320–339.

[13] D. Kebbal. 2006. Automatic flow analysis using symbolic execution and path enumeration. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*.

[14] Tien-Duy B Le, Xuan-Bach D Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing specification miners through model fissions and fusions (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 115–125.

[15] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL specification mining (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 81–92.

[16] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2–11.

[17] David Lo and Siau-Cheng Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*.

[18] David Lo, Siau-Cheng Khoo, and Chao Liu. 2008. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 20, 4 (2008), 227–247.

[19] R. Nelken and N. Francez. 1996. Automatic translation of natural language system specifications into temporal logic. In *International Conference on Computer Aided Verification*. 360–371.

[20] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. 2012. GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool. In *2012 34th International Conference on Software Engineering (ICSE)*. 1407–1410. DOI:http://dx.doi.org/10.1109/ICSE.2012.6227236

[21] Tung Nguyen, Hoan Nguyen, Nam H. Pham, Jafar Al-kofahi, and Tien N. Nguyen. 2009. Graph-based mining of multiple object usage patterns. 383–392. DOI:http://dx.doi.org/10.1145/1595696.1595767

[22] Michael Pradel and Thomas R Gross. 2009. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 371–382.

[23] Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26, 6 (2009), 27–34.

[24] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (2013), 613–637.

[25] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 598–608. http://dl.acm.org/citation.cfm?id=2818754.2818828

[26] Mohamed Aymen Saied, Hani Abdeen, Omar Benomar, and Houari Sahraoui. 2015. Could we infer unordered API usage patterns only using the library source code?. In *International Conference on Program Comprehension (ICPC)*. IEEE Press, 71–81.

[27] Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. 2015. Mining multi-level API usage patterns. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 23–32.

[28] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. 2018. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software* 145 (2018), 164–179.

[29] Mohamed Aymen Saied and Houari Sahraoui. 2016. A cooperative approach for combining client-based and library-based API usage pattern mining. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 1–10.

[30] Mohamed Aymen Saied, Houari Sahraoui, Edouard Batot, Michalis Famelis, and Pierre-Olivier Talbot. 2018. Towards the automated recovery of complex temporal API-usage patterns. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 1435–1442.

[31] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. 2015. An observational study on API usage constraints and their documentation. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 33–42.

[32] Harold Somers. 1999. Review Article: Example-based Machine Translation. *Machine Translation* 14, 2 (1999), 113–157.

[33] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *36th International Conference on Software Engineering, ICSE '14*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 643–652.

[34] Xiaobing Sun, Bin Li, Yucong Duan, Wei Shi, and Xiangyue Liu. 2016. Mining Software Repositories for Automatic Interface Recommendation. *Sci. Program.* 2016 (June 2016), 5–.

[35] Gias Uddin, Barthélémy Dagenais, and Martin P Robillard. 2012. Temporal analysis of API usage concepts. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 804–814.

[36] Jianyong Wang and Jiawei Han. 2004. BIDE: Efficient mining of frequent closed sequences. In *International Conference on Data Engineering*. 79–90.

[37] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*. ACM, 282–291.

[38] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *European Conference on Object-Oriented Programming*. 318–343.

[39] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, 318–343.

[40] Zixiao Zhu, Yanzhen Zou, Bing Xie, Yong Jin, Zeqi Lin, and Lu Zhang. 2014. Mining api usage examples from test code. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 301–310.