Towards the Automated Recovery of Complex Temporal API-Usage Patterns

Mohamed Aymen Saied
Concordia University
m_saied@encs.concordia.ca

Houari Sahraoui
Université de Montréal
sahraouh@iro.umontreal.ca

Edouard Batot
Université de Montréal
batotedo@iro.umontreal.ca

Michalis Famelis
Université de Montréal
famelis@iro.umontreal.ca

Pierre-Olivier Talbot
Université de Montréal
po.talbot1@gmail.com

ABSTRACT

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 understand and use those libraries. However, most of these techniques produce basic patterns that generally do not involve temporal properties. In this paper, we discuss the problem of temporal usage patterns recovery and propose a genetic-programming algorithm to solve it. Our evaluation on different APIs shows that the proposed algorithm allows to derive non-trivial temporal usage patterns that are useful and generalizable to new API clients.

ACM Reference Format:
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 2018 (GECCO '18).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In modern software development, to achieve any meaningful task of non-trivial complexity, developers need to 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 are 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 [8]. 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.

GECCO '18, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Recently, much research effort has been dedicated to the identification of sequential API usage patterns (ordered sets of co-used methods) [11]. Other research contributions targeted unordered API usage patterns [4, 10]. Given a set of client programs that use a considered library, existing techniques identify API usage patterns that are recurrent in the given set of clients. The inferred usage patterns with these techniques tend to be many, redundant and simple, posing significant barriers to their practical usefulness with tasks intended for experienced developers, such as ensuring that the software remains correct while features are added or removed.

However, mining temporal aspects of such constraints, i.e., latent temporal properties is still an open question. Some work has been done on mining temporal specifications from libraries in the form of automata [6] or rules [4, 7]. Others have attempted to uncover latent behaviour in UML models [5]. 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. [2]. 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 approach 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 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.

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. We conclude this paper in Section 6.

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 [2]. 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 an 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 are 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 the mining of API specifications and usage patterns [9]. 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 [9].

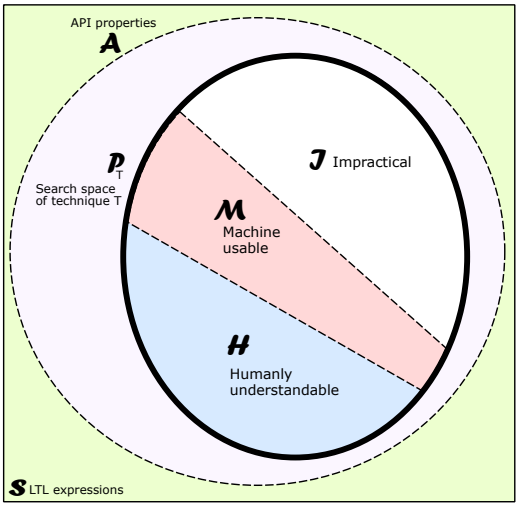

Figure 1: Overview of the search space.

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. [13] 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. [14] 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. [10] 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 [12]. 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 [3], 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 [7]. 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. [11] 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.* [1] 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 [1].

4 COMPLEX TEMPORAL API USAGE PATTERNS MINING

Our approach consists in using genetic programming, an evolutionary method, to mine temporal patterns from execution traces. 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. We describe in this section the adaptation of genetic programming (GP) to learn temporal API usage patterns. To apply GP to a specific problem, one must specify the encoding of solutions, the operators that allow movement in the search space so that new solutions are obtained, and the fitness function to evaluate a solution's quality. These three elements are detailed in the next subsections.

4.1 Overview

Figure 2 provides a high-level overview of the 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 until the algorithm performs a certain number of iterations as an end criteria. In the remainder of this section, we give the details of our algorithm.

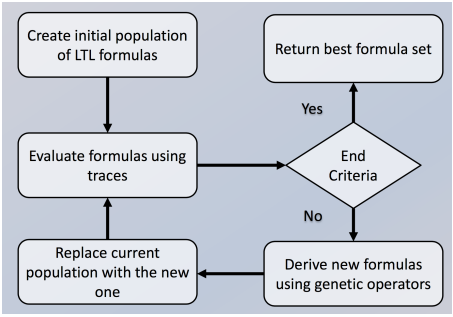

Figure 2: TApproach overview

4.2 Genetic Programming

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. A program (an individual of the population) is usually represented in the form of a tree, where the internal nodes are functions (operators) and the leaf nodes are terminal symbols. Both the function set and the terminal set must contain symbols that are appropriate for the target problem.

Each individual of the population is evaluated by a fitness function that determines its ability to solve the target problem. Then, it is subjected to the action of genetic operators such as reproduction and crossover. The reproduction operator selects individuals in the current population in proportion to their fitness values, so that the more fit an individual is, the higher the probability that it will take part in the next generation of individuals. The crossover operator replaces a randomly selected subtree of an individual with a randomly chosen subtree from another individual.

Once reproduction and crossover have been applied according to given probabilities, the newly created generation of individuals is evaluated by the fitness function. This process is repeated iteratively, usually for a fixed number of generations. The result of genetic programming (the best solution found) is the fittest individual produced along all generations.

The following subsections describe our adaption of GP to the API temporal patterns mining problem.

4.3 Modelling LTL Patterns

LTL patterns are represented as an unbalanced binary tree. Every node of the tree represents an element of the pattern. The tree nodes can be one of two types: Operator nodes or Variable nodes, with leaf nodes representing variables and non-leaf nodes representing operators. The tree is built so that the subtree(s) of a given node represents that operator's operand(s).

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

- Operator nodes must contain one operator accepted in LTL syntax and have at least one child node. The internal representation of the node's operator contains its name, symbol and type (Unary or Binary).
- Variable nodes must contain a variable (a single character that is not reserved by the LTL syntax) and no child nodes. Variable nodes are bound to the library methods called in the traces.

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.

Figure4 shows an example of such a tree. The LTL pattern represented by this tree 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() with the restriction that method m2() cannot be called before the method m1()

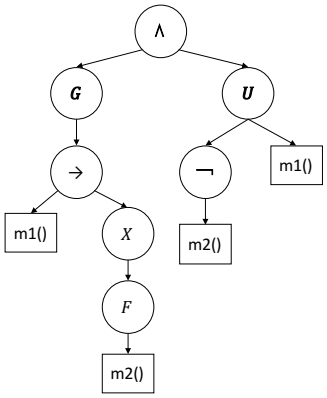

Figure 3: LTL binary tree

4.4 Initial Pattern Set Generation

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' 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.

4.5 Fitness function

To evaluate the fitness of LTL patterns, we have to evaluate the "performance" of each pattern on the execution trace. To do so, GenLTL uses the *texada* tool [1]. This tool (with specific parameters) takes an LTL and a trace as parameters and returns all possible bindings of said LTL given the trace. These bindings are evaluated through three metrics: Support, Potential Support, and Confidence. We use these three metrics to assign a fitness score to every generated LTL pattern.

Support Potential: For a pattern p, support potential is the number of events in the trace that could falsify p.

Support: For a pattern p, support is the number of events that could falsify p, but do not falsify p.

Confidence: For a pattern p, confidence is the ratio of support over support potential.

In other words, support potential represent the degree to which a particular pattern *could* apply to a trace, support the degree to which it actually does. The confidence metric exposes the relationship between the two.

The fitness function defined in defined below should reflect the following constraints:

- 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.
- Penalize patterns that over support. Patterns with extremely high support are so general, that they are usually either trivial or not worthy of interest.

For these reasons, The fitness function should separate the patterns in three "layers". The bottom layer for 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 boosts for the over-support and under-support category is 0, and for good-support category is 1/2.

We therefore define the fitness function of GenLTL as:

$$fitness = \frac{1}{2}FitnessBoost + \frac{1}{2}Confidence \times \frac{SupportPotential}{TraceLength}$$

The under-support limit is fixed at 3 whereas the over-support limit is the length of the trace. Once every pattern is generated and scored, they are ranked in descending order of score.

4.6 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 proprety-checking. In order to conserve integrety 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 The crossover operator take 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 operaotrs. 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 becomming 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.

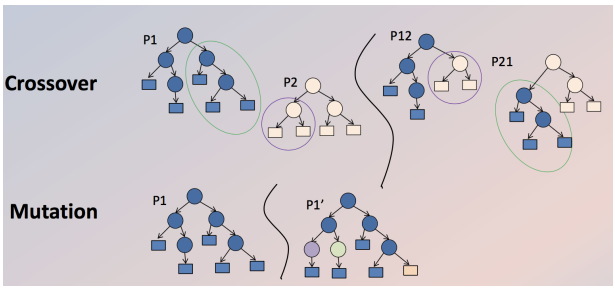

Figure 4: Genetic Operators

5 EVALUATION

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

- **RQ1:** What kind of patterns we can interfere 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?

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. The APIs usage was inferred from the execution trace of typical usage scenarios for each app. We considered the methods in the execution traces at the first nesting level on the library side, to only keep the call to API methods from client methods.

For each experiment in this section, we present the research question to answer, the research method to address it, followed by the obtained results.

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

To address (RQ1) we run GenLTL on the whole dataset of traces over a horizon of 50 generations with a population size of 300. To evaluate the structure of the inferred patterns and how complex they are, we compute the following metrics presented in Table 5. Number of Events is he 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.

Table 2: 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	$G(x \rightarrow Xy) \vee G(x \rightarrow Xz), G(a \rightarrow XF(z)) \vee G(z \rightarrow Xw)$	$G(a \rightarrow XF(z)) \vee G(x \rightarrow Xz), G(x \rightarrow Xy) \vee G(y \rightarrow Xw)$

Table 3: Descriptive statistics of our setting and results

	Traces		Patterns					
API	Nb events	Nb Traces	coverage	Nb patterns	Nb methodPerPattern	depthPattern	widthPattern	support
database	4057,00	10,00	0,44	124,00	6,07	2,40	7,00	567,47
graphics	1210,00	21,00	0,08	115,00	5,26	2,23	6,34	278,05
hardware	559,00	3,00	0,47	80,00	3,33	2,42	6,76	158,13
text	483,00	12,00	0,21	89,00	3,87	2,37	6,99	101,28
util	878,00	27,00	0,08	115,00	5,36	2,34	6,74	168,64
view	1255,00	45,00	0,12	115,00	5,15	2,24	6,59	335,73
webkit	320,00	15,00	0,33	110,00	4,78	2,51	7,09	28,47
widget	444,00	5,00	0,31	94,00	3,99	2,23	6,45	86,40

A cursory look at the results of table 5 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 proper. If, by oposition, 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. 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 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 that those found in other APIs. Indeed, the number of methods per pattern, the depth and width of patterns are all higher in this API as compared to others. This fact seems to indicate that a bigger number of methods are interdependant 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?*

The mining of usage patterns for an API depends on the used set of API's client programs (training client programs). Hence, to address our second 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.

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 wich confidence degree. the goal is to see if good patterns remain good in the context of new client programs.

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 wors degradation for was the widget API, the confidence degree remain over 90 % in both training and validation context.

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?*

To 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. 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 then analyzed 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 4. P_1 is an exclusive disjunction (XOR) decomposition of two base block subformulas. The first subformula describes the constraint that an update operation 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.

Table 4: 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.

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 4. 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.

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 [1] 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.

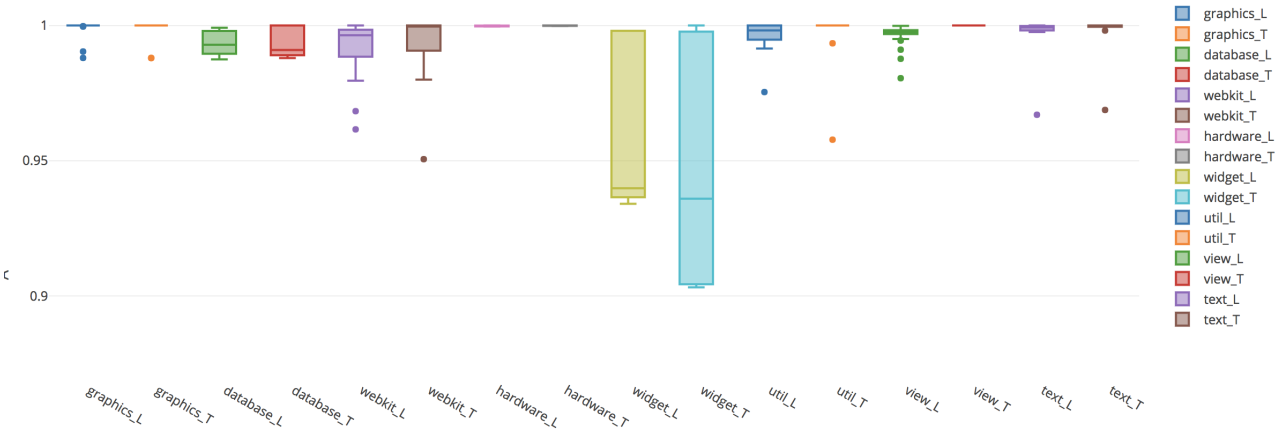

Figure 5: Patterns Generalizability

6 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 applicability of candidate patterns to the trace samples. Unlike most of the existing approaches, our 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 uncommon usage situations.

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.

REFERENCES

[1] Caroline Lemieux Dennis Park Ivan Beschastnikh. General LTL Specification Mining. (????).
[2] 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
[3] 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.
[4] 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.
[5] 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
[6] 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*.
[7] 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.
[8] Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26, 6 (2009), 27–34.
[9] 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.
[10] 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.
[11] Gias Uddin, Barthélémy Dagenais, and Martin P. Robillard. Temporal Analysis of API Usage Concepts. In *International Conf. on Software Engineering*.
[12] 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.
[13] 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.
[14] 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.