# Checking temporal patterns of API usage without code execution

Erick Raelijohn
Université de Montréal
Montréal, Canada
erick.raelijohn@umontreal.ca

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

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

Abstract—In theory, Application Programming Interfaces (APIs) allow developers to write client code that reuses library code without needing to know its internals. In practice, correctly using APIs requires respecting explicit and implicit constraints and usage temporal patterns. Ignoring such patterns could lead to errors and design flaws. These often cannot be detected prior to integration and system testing. We propose the Temporal Usage PAttern Checker (Tupac) for APIs, an interprocedural static analysis approach that can verify that client code conforms to temporal API usage patterns as it is being developed. We evaluated the effectiveness of our approach on 4 projects with 4 different APIs. Our evaluation shows that Tupac allows to accurately check the conformance of the client code to the patterns in under 1 second. This means that Tupac can realistically be deployed in an IDE without stalling the regular coding rhythm.

## I. Introduction

Reuse is a fundamental concept in software engineering. In contemporary practice, it frequently takes the form of using software libraries and frameworks via an Application Programming Interface (API). An API exposes a series of functionality points, enabling client code to make use of the library without needing to know its internals. Despite its advantages, using APIs is not a straightforward task. Effective use of libraries is rarely limited to using a single API call. Consider an API for writing to a resource: a developer would need to open() it, write() to it, flush() the writer, and close() the resource. A task described by a single verb ("write") requires in fact the correct combination of four API calls (open, write, flush, close). Appropriately performing such combinations is cognitively challenging. This is exacerbated if by method name overloading, whereby an API method could be defined for multiple parameter lists.

Learning how to use an API requires considerable investment in time and effort [16]. In addition to reading and understanding documentation, developers also rely on community support such as examples found online. This is because APIs are more than the sum of their methods: they are complex tools, meant to be used according to specific usage scenarios and best practices. These are documented at various degrees of quality by API publishers. To complete the documentation, several researchers have focused on uncovering such intended scenarios and practices from publicly available client code in the form of API usage patterns. Roughly speaking, API usage patterns can be seen as groups of API methods that are called in a coordinated way in the client code [30]. When those groups include temporal properties such as calling order, they are referred to as temporal patterns [22].

However, whether API usage patterns were produced manually or recovered automatically it is not enough to just record them in API documentation documents. Instead, they can be made useful in practice, helping developers check their code for conformance or guiding them when using the API. In contemporary practice, developers typically rely on modern Integrated Development Environments (IDEs) to ease their cognitive load during their work. This can take the form of, e.g., contextual help that shows a part of the documentation on the fly, code completion to help improve programming productivity, or real time checking of code syntax. Such features are implemented using static analysis techniques and are generally provided locally, at the level of a method or an individual line of code. Thus, they are not usually able to handle more complex properties such as temporal API usage patterns. Checking temporal properties interprocedurally (i.e., across different methods of a code project) requires dynamic analysis by executing the code and analyzing traces. However, this cannot be done without writing test cases, which breaks the natural rhythm of coding, as it requires writing glue code and/or test fixtures. Even more, it cannot be done for code that is in the process of being written, as such code is by definition incomplete.

In this paper, we aim to bridge the gap between research on the recovery and documentation of temporal API usage patterns [30], [14], [16], [15], [22], [29], [12], [23] and its application in practice in contemporary IDEs. We want to leverage the patterns to increase developers' productivity by helping them make better use of APIs in their regular coding rhythm. We thus introduce the Temporal Usage PAttern Checker (Tupac) for APIs, an approach that leverages knowledge about API usage patterns directly inside the IDE, during development. Given a set of API usage patterns, Tupac statically analyzes client code as it is written, and uses a model checker to generate direct feedback about API usage to the developer. Tupac has

Listing 1: Original close() method that causes an error.

```
1  public class DataManager extends SQLiteClosable {
2    ...
3    Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
4    public void close() {
5      Iterator<Entry<SQLiteClosable, Object>> iter =
           mPrograms.entrySet().iterator();
6      while (iter.hasNext()) {
7        Map.Entry<SQLiteClosable, Object> entry = iter.next();
8        SQLiteClosable program = entry.getKey();
9        if (program != null)
10         program.onAllReferencesReleasedFromContainer();
11 }}}
```

Listing 2: Corrected version of the close() method.

```
1  public class DataManager extends SQLiteClosable {
2    ...
3    public void safeClose() {
4      Iterator<Entry<SQLiteClosable, Object>> iter =
           mPrograms.entrySet().iterator();
5      this.safeRelease();
6      while (iter.hasNext()) {
7        Map.Entry<SQLiteClosable, Object> entry = iter.next();
8        SQLiteClosable program = entry.getKey();
9        if (program != null)
10         program.onAllReferencesReleasedFromContainer();
11     }
12     onAllReferencesReleased();
13   }}
14   public void safeRelease(){
15     for (SQLiteCompiledSql compiledSql : this.CQueries.values())
16       compiledSql.releaseSqlStatement();
17     CQueries.clear();
18 }}
```

been implemented as an Eclipse[1] plugin, which will be made available upon publication.

We conducted an empirical evaluation of the correctness and scalability of Tupac. We found that, despite not using dynamic analysis, Tupac is able to detect patterns violations with a good precision and recall. In terms of performance, we observed that Tupac takes half a second on average to check a pattern for a whole project. The key contributions of this paper are: (1) A technique that uses static interprocedural analysis to capture possible executions of client code that uses an API. The interprocedural analysis relies on a Deep Graph that combines the client's call graph with the control flow graphs of its methods. (2) An approach to model check temporal patterns on a finite state automaton, derived from the client's Deep Graph. (3) An Eclipse plugin, Tupac, that helps developers leverage API temporal patterns while they are programming.

The rest of the paper is organized as follows: We provide a motivating example in Section II. We describe background and related work in Section III. We introduce formal definitions in Section IV and our approach in Section V. We present an evaluation of our approach in Section VI and conclude in Section VII.

## II. Motivating Example

To better motivate and illustrate our proposed approach, we consider the scenario of a developer named Dev who is working on an mobile app that requires persistent storage. Dev is working on the class DataManager that uses the Android SQLite API to manage the app's database connection but he is not experienced with this API. We show a snippet of Dev's class DataManager in Listing 1. The class DataManager extends the API class SQLiteClosable and on line 10 its method close() calls the API method onAllReferencesReleasedFromContainer(). As indicated in the documentation[2], onAllReferenceReleasedFromContainer() is used to finalize the object, i.e., to release all references to it and close the database if the last connection is removed. If Dev was more familiar with the API, he would know that an error is caused if the database is closed before SQL statements are finalized. This is exactly what happens in DataManager because the object CQueries is initialized in line 3 but is not finalized before line 10. As Dev is not experienced, he is unlikely to write a test that would uncover the problem.

A more experienced developer might de-allocate all compiled SQL statements before closing the connection to the database. From experience, they would know that to avoid the error, it is necessary to call the method releaseSqlStatement() before the reference to SQLiteClosable is released. This pattern is followed in the corrected version of the close() method, safeClose(), in Listing 2. In this version, before calling onAllReferenceReleasedFromContainer(), we invoke the helper method safeRelease() in line 4, which safely releases the compiled SQL statements (lines 14-17).

Such usage patterns are examples of best practices for using APIs. There exist several techniques [24], [23], [7] that can automatically mine such API usage patterns, rules, or guidelines from publicly available code. In our example, the rule about de-allocating compiled SQL statements before closing the database can be captured by the Precedence property pattern in the "globally" scope [6] and represented as the Linear Temporal Logic (LTL) formula $\phi_1 = \neg a \; W \; rs$ where the variable a stands for the method call SQLiteClosable#onAllReferencesReleasedFromContainer(); the variable rs for the call SQLiteCompiledSql#releaseSqlStatement(); and $W$ is the "weak until" LTL operator [8]. Such a pattern could have been mined by one of the aforementioned techniques, or written by hand as documentation.

Our goal is to leverage patterns such as $\phi_1$, regardless of whether they were mined automatically or documented manually, to help Dev overcome his inexperience with the SQLite API. This could be done in various ways, such as by supporting him in writing better test cases, by dynamic analysis, or by recommending documentation and relevant code snippets. Here, we focus on Dev's immediate coding experience: can we provide him with timely feedback on whether his code respects $\phi_1$ right in his IDE? And can we generate this kind of feedback even when the code is not complete or cannot be fully executed?

Our basic idea is to perform static analysis on Dev's client code. The Control Flow Graph (CFG) of the method

[1] https://www.eclipse.org/

[2] https://developer.android.com/reference/android/database/sqlite/SQLiteClosable

Datamanager.close(), shown in Figure 1, is a good starting place if we can use an off-the-shelf model checker, such as NuSMV [4] to find an execution path that violates $\phi_1$. Finding such a path indicates that there is a risk that once the code is executed, the pattern would be violated. It can easily be observed that the pattern is not respected: there is no path containing $rs$ i.e., all paths on the CFG violate $\phi_1$.

Given the findings of the model checker, we could help Dev in multiple ways. We can inform him that his client code violates an API usage pattern, and show him graphically the violating execution path found by the model checker. This would guide him in finding the problem in his code and correcting it. The graphical visualization of the path might also help detect any unexpected behaviours and thus show Dev how he should extend the coverage of his tests. In such a case, the results of static analysis could help improve the set up of dynamic analysis. In short, giving Dev this feedback directly in the IDE could prompt him to further debug and correct his implementation. For this to work, the static analysis must not take too long, otherwise Dev might just turn it off, as it would interrupt his coding rhythm.

But what about safeClose(), shown in Listing 2? Its CFG does not contain the call to releaseSQLStatement() because it is in the helper method safeRelease(). In other words, we need a more sophisticated analysis across different methods. In the following, we describe Tupac, a technique that supports both intraprocedural (within a single method) and interprocedural (across multiple methods) static analysis of temporal API usage patterns, within the constraints of needing to operate embedded in the regular rhythm of coding in an IDE.

## III. Background and Related Work

Developers frequently use stylized sequences of instructions to use APIs. We call these stylized sequences API usage patterns. Consider how we use the java#util#ListIterator API: regardless of the particular application scenario, we typically invoke hasNext() first to check if there is an element remaining to iterate and then move the cursor using next(). Java's "for-each" syntactic construct can guide developers in using this basic pattern. However, other pattern variations can exist; below we call these alternatives. In the ListIterator example, we have different alternatives depending on the type of loop (while/for) or the order of iteration (e.g., using hasPrevious() and previous()). In practice, finding usage patterns requires a lot of effort from developers. This is because API documentation is typically given at the level of individual calls. In the ListIterator's documentation[3] the usage pattern described above is not described in neither the top-level documentation of the interface nor the documentation of hasNext() or next(). It is also not documented at the documentation of the parent interface Iterator. Instead, developers have to rely on finding examples of API usage.

Usage patterns can be very useful when developing systems with APIs. On the one hand, the patterns can be leveraged to recommend API call sequences to help achieve a particular task. This makes it easier to both learn and use the APIs correctly, especially since discovering the subset of an API that is relevant to a given task poses a big challenge [20]. On the other hand, if we assume that the patterns represent correct API usage, we can use them to find uncommon usages of the API in client. This can help during development phases such as quality assurance (e.g., with code reviews) and system maintenance (e.g., during debugging).

Discovering API usage patterns highly non-trivial, requiring a lot of effort from developers. Researchers have thus tried different techniques for automatically recovering API usage patterns. Zhong et al. [30] introduced MAPO, a framework for mining API usage patterns from open-source repositories. Nam et al. [14] presented the usage pattern mining algorithm MARBLE, that relies on pattern boilerplates to ease the detection of certain API usability issues. FOCUS is a recommender system proposed by Nguyen et al. [16]. It can extract API patterns and generate recommendations to developers to help them find suitable API function calls and usage patterns. Nguyen et al. [15] proposed a context sensitive code completion tool based on templates of frequent API usage patterns. Saied et al. [22] proposed to automatically identify collections of libraries that are commonly used together to recover patterns. Another popular approach is to mine temporal logic patterns from client code. Yang et al. [29] use simple property templates such as "$\text{event}_0$ always followed by $\text{event}_1$", based on the Response pattern [6]. Lo et al. [12] proposed a multi event approach that uses the template "whenever a series of events occur, eventually, other series of events will occur". Saied et al. [23] proposed an approach that avoids using templates to learn complex, non-trivial usage patterns using genetic programming.

In this paper, we take such pattern recovery techniques as a given. We do not make any assumptions about the provenance of API usage patterns: they could have been mined or manually composed. Instead, we focus helping developers leverage the patterns to better use APIs in their regular coding rhythm. The only assumption that we make is that the patterns are expressed in temporal logic, and specifically Linear Temporal Logic (LTL) [8]. Several of the aforementioned approaches (e.g., [23]) output the patterns they recover to LTL, or can be adapted to do so.

Providing useful information to developers is an active research area. Work by Rastkar et al. [19] and Murphy in [13] highlights the role of IDEs in assisting developers in their work by reliably providing them with knowledge according to their task specific needs. Similarly Robillard et al. [21] focus on generating context sensitive knowledge on demand to the developer. Kersten et al. [9] filter

[3] docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html

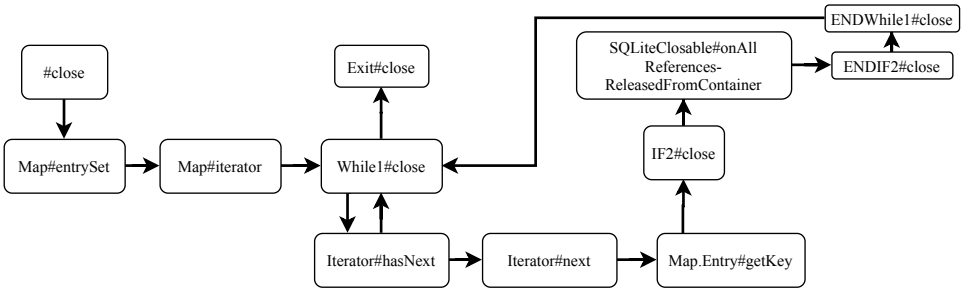


Fig. 1: Control Flow Graph (CFG) of the method close().

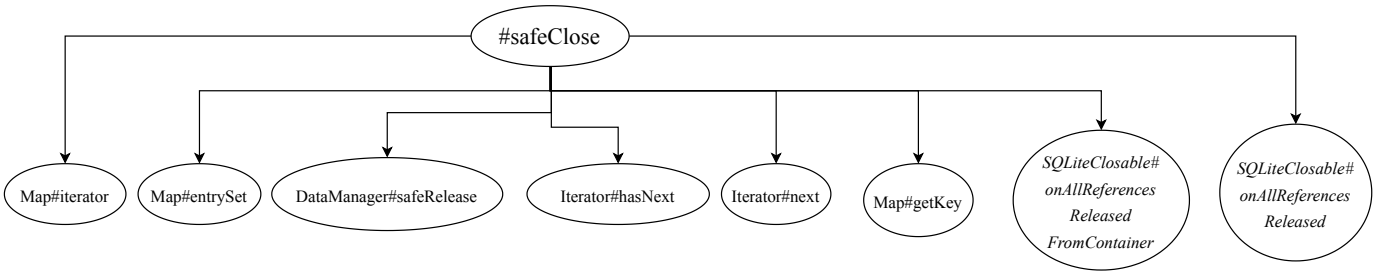


Fig. 2: Call Graph (CG) of the method safeClose() from Listing 2

information provided by the IDE to avoid overflow of information to the developer that can hinder productivity. On the same basis but focusing more on API usage, Xu et al. [28] introduced MULAPI, a tool that recommends APIs from feature request documents. Raelijohn et al. [18] proposed a vision to help client application developers to correctly use an API at different development phases using temporal API usage patterns.

Other approaches focus on helping detect API misuse. Wen et al. [27] propose MutAPI, a tool for detecting API misuse patterns. Using mutation analysis, Amann et al. [2] introduce MuDETECT an API misuse detector that depends on graph structures to map API usage. While several researchers have focused on recovering API usage patterns, few have studied how to use the recovered patterns in patterns. This is also evidenced by the lack of an ecosystem of IDE plugins.

Using static analysis for detecting API misuse is an active research area [3]. Wasylkowski et al. [26] developed Jadet, an approach that uses an intraprocedural analysis to detect patterns in the form of sets of pairs of method calls. They also proposed Tikanga [25], an approach to mine temporal patterns on methods' parameters (preconditions). They mention however that Tikanga does not apply to API usage. Nguyen et al. [17] proposed an approach based on the creation of graph representations of code, called "groums". In fact, their tool Grapacc [15], is the closest to Tupac. It is an IDE plugin that statically analyzes client code and then generates recommendations for developers by finding similarities between extracted groums and a database of known usage pattern groums. Tupac differs from these three approaches in two ways: (a) it can handle patterns of arbitrary complexity, and is not based on pairs or sequences of method calls or preconditions); (b) it can perform interprocedural analysis. It works by analyzing client code to verify its compliance with a set of given patterns. This can uncover API misuses or confirm the good application of the patterns. To do this, we take inspiration from the work of Dagenais and Hendren [5], to extract information from the client code using static analysis and structure it in a graph-based intermediate representation model.

In Tupac, an API usage pattern is a given LTL formula [8], where variables represent API calls. An example of a usage pattern written in LTL is the pattern $\phi_1$, described in Section II. These patterns are evaluated on paths over a graph structure (called "Deep Graph", see the following section) that integrates information from the call graph and the control flow graph of client code.

## IV. Preliminaries

In this section, we describe the code abstractions used by Tupac to represent software projects. We assume an object oriented paradigm, under which a software project consists of a set of classes, each of which contains a number of methods, each of which contains an ordered set of instructions. We denote the set of all methods of a system as $\mathcal{M}$, and the set containing the union of all instructions of all methods as $\mathcal{I}$. For simplicity, in the following we assume that chained instructions, such as the instruction mPrograms.entrySet().iterator() in line 5 of Listing 1 are unchained (e.g., as x=mPrograms.entrySet(); i=x.iterator()).

One abstraction that Tupac uses is the Control Flow Graph (CFG), a model of computer programs that allows

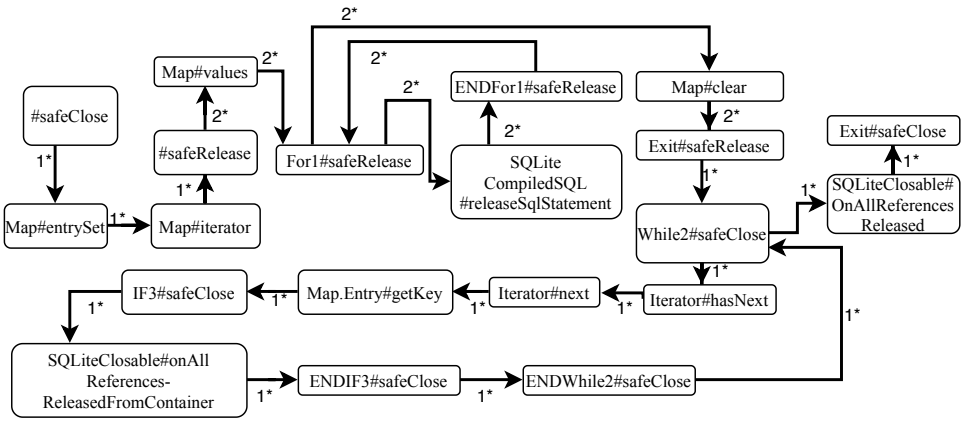


Fig. 3: Deep graph of the method safeClose() from Listing 2.

visualizing possible execution traces [1]. Here, we use the following definition:

Definition 1 (Control Flow Graph (CFG)): The CFG of a method $K$ that has no branches in except for entry and no branches out except at the exit is a tuple $\langle M, M_0, M_f, F, n_{CFG} \rangle$ where: $M$ is a set of nodes representing the instructions of $K$; $M_0$ is the initial node $M_0 \in M$, representing the entry point of $K$; $M_f$ is the set of exit nodes $M_f \subseteq M$, representing the exit points of $K$; $F$ is a set of directed edges such that $F \subseteq M \times M$ representing the flow of control in $K$; and $n_{CFG}$ is a naming function $n_{CFG} : M \to \mathcal{I}$ that labels each node with the instruction it represents.

Conditional statements, loops, and try-catch statements, are represented as nodes in the CFG that have more than one outgoing control edges. They are called predicate nodes. For simplicity of presentation, we assume that each predicate node is paired with a corresponding "end" pseudonode (e.g., an if is paired with an endif, and so on). For example, the CFG of the method DataManager.close() from Listing 1 is shown in Figure 1, where nodes are labelled by the corresponding instructions in Listing 1. The set $M$ of this CFG is the set of all nodes; and $M_0$ is the node Map#entrySet ; $M_f$ is the set that contains node while1#close; and the set $E$ is the set of all arrows. Any path on this CFG represents a possible execution trace of the method.

The CFG allows some forms of intraprocedural analysis (within a single method) but does not provide any interprocedural (across methods) information. To represent interprocedural calls, we use a particular kind of control flow graph, the Call Graph (CG), defined as follows:

Definition 2 (Call Graph (CG)): The CG of a system is a directed graph defined as a tuple $\langle V, V_e, E, n_{CG} \rangle$, where: $V$ is a set of nodes representing methods of classes of the system; $V_e$ is a node in $V$ representing the execution entry point of the system; $E$ is the set of edges representing caller-callee relationships $\langle V_i, V_j \rangle$ between the methods represented by $V$; and $n_{CG}$ is a naming function $n_{CG} : V \to \mathcal{M}$ that labels each node with the method it represents.

We say that the CG is "centred" on the method $V_e$. We assume that $E$ only contains caller-callee relationships that respect any relevant visibility rules. We scope our approach to static analysis and we thus do not consider polymorphic calls, code injection, etc. Given a particular subset of methods of interest of the system and a given call depth $d$, we can always construct a finite slice by excluding all method calls that are not of interest or that cannot be reached from $V_e$ in $d$ steps. In the following, when talking about CGs, we always refer to such relevant finite slices. For example, the CG of Dev's app consists of the methods of its classes, as well as the API methods that it uses. Assuming that Dev is using the corrected version of close() from Listing 2, we show a slice of this CG that centered on the method safeClose() in Figure 2. In this slice, the safeClose() method is the entry point $V_e$, and the set $V$ only contains the methods of the class DataManager and calls to the Android SQLite API. In the figure, we indicate the caller-callee relationships as arrows. For example, since the method safeClose() calls the API method SQLiteCloseable.onAllReferencesReleasedFromContainer() the two corresponding nodes are connected via an arrow.

While the CG visualizes interprocedural dependencies between methods, it does not expose the internal workings of each called method. In order to analyze both, we use a hybrid of the CFG and CG, which we call "Deep Graph" (DG). The DG combines relevant information from the CFG and CG to allow interprocedural analysis of API usage. We detail the construction of the DG in Section V. Intuitively, the DG uses the CG of a system to link together the CFGs of the methods. More specifically, we define it as follows:

Definition 3 (Deep graph): The DG of a system is a directed graph defined as a tuple: $\langle N, N_0, \Delta, n_{DG} \rangle$, where: $N$ is a set of nodes representing instructions in the methods of the system; $N_0$ is a node $N_0 \in N$

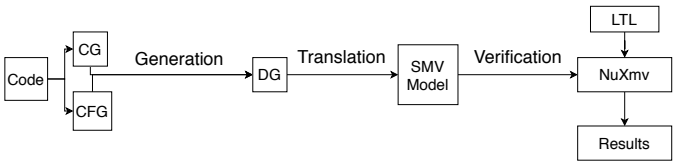


Fig. 4: Approach overview

representing the entry point of the system; $\Delta$ is a set of edges $\Delta \subseteq N \times \mathcal{M} \times N$ that labels each flow of control with the method in $\mathcal{M}$ that triggered it; and $n_{DG}$ is a naming function $n_{DG} : N \to \mathcal{I}$ that labels each node with the instruction it represents.

The DG of the method safeClose() from Listing 2 is shown in Figure 3. It combines the CFGs of safeClose() and safeRelease() in a single representation, using the caller-callee relationships described in the CG slice centred around safeClose() that we show in Figure 2. The CG also shows the caller-callee relationships between other elements, e.g., SafeClose() and Iterator.hasNext(). Since the latter is not an API call or a client code method, we have not expanded its CFG into the DG. Instead, it appears as a regular node in the DG. On the contrary, we have expanded the CFG of safeRelease() in the DG since it is a client code method and therefore could contain API calls.

A path on a CFG, a CG, or a DG, represents a possible flow of control in the system. We take a static analysis perspective aiming to reason about systems without executing them. Thus, we call these execution paths "traces", even though they do not correspond to actual execution traces. They represent potential executions of the system and so can offer insights about correct API usage. Formally, we define a trace as a set of possible executions of a program $P$ given an entry point:

Definition 4 (Trace): A trace is a sequence $T = \langle t_0, t_1, ..., t_n \rangle$ of nodes $t_i$ of the DG that starts from an entry point $V_0$ and ends at a node that has no outgoing edge $V_x$.

The challenge is to find such traces and to check whether each of them satisfy a given LTL specification. Typically, this is defined as a model checking problem [8]. For Tupac, we express the DG of a system in a representation that can be processed by a symbolic model verifier (SMV), following the NuSMV approach [4]:

Definition 5 (SMV model): An SMV model is state transition model (a Kripke structure [10]), defined as the tuple $M = \langle S, I, R, L \rangle$ where, $S$ is a finite set of states, $I$ is a set of inital states such that $I \subseteq S$, $R$ is a transition relation such that $R \subseteq S \times S$ and $L$ is a labelling of $S$ such that $L : S \to 2^{P\mathcal{I}}$, where $P\mathcal{I}$ is the set of boolean expressions over the set $\mathcal{I}$ of system instructions.

SMV models are represented in the NuSMV language [4]. Given an SMV model $M$ and an LTL formula $\phi$ the model checker finds the set of all states that satisfy $\phi$ namely, $s \in S \cdot M, s \vdash \phi$. A program satisfies $\phi$ if all $s$ satisfy $\phi$. In the case where $s_o \not\models \phi$, $s_o$ is considered as a counter-example, i.e., an execution trace that violates $\phi$.

**Algorithm 1** ConstructDG: Deep Graph generation

Inputs
  Set of CFGs: $S_{CFG}$ : {cfg : $\langle M, M_0, M_f, F, n_{CFG} \rangle$}
  Call graph: cg:$\langle V, V_e, E, n_{CG} \rangle$
Output
  Deep graph: $dg : \langle N, N_0, \Delta, n_{DG} \rangle$
A) Initialization
$g$ = getCFG($V_e$).$M_0$
prune($g$)
$N_0 = g$; $N = N_0$; $\Delta = \emptyset$; $n_{dg} = \emptyset$
B) Build forest of annotated CFGs
for each $v \in V - N_0$ do
  c = getCFG(v)
  prune(c)
  $N := N \cup c.M$
  $n_{dg} := n_{dg} \cup c.n_{cfg}$
  for each edge $s \to t \in c.F$ do
    $\Delta := \Delta \cup s \xrightarrow{v} t$
C) Link the forest of CFGs
for each cfg $y \in S_{CFG}$ do
  for each $s \to t \in y.F$ do
    $m_s = y.n_{CFG}(s)$
    $m_c = getCFG(m_s)$
    if $m_c = \emptyset$ then continue ▷ i.e. s is a predicate or an API call
    if $m_c = y$ then continue ▷ we ignore recursion
    $l = y.n_{CFG}(y.M_0)$
    $\Delta := \Delta \cup s \xrightarrow{l} m_c.M_0$
    $\Delta := \Delta \cup m_c.M_f \xrightarrow{l} t$
    $\Delta := \Delta - (s \xrightarrow{l} t)$
return dg:$\langle N, N_0, \Delta, n_{dg} \rangle$
procedure getCFG($v \in V$)
  for each cfg $y \in S_{CFG}$ do
    if $n_{CG}(v) = y.n_{CFG}(y.M_0)$ then
      return $y$
  return $\emptyset$

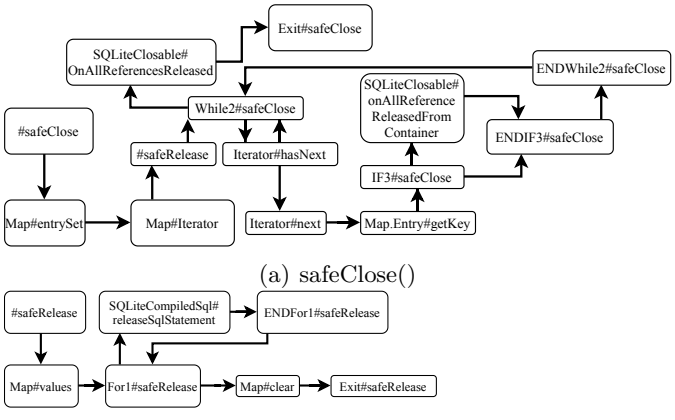


Fig. 5: CFG of methods in Listing 2.

## V. Approach

In this section, we describe in more detail our algorithm for the generation of the Deep Graph and its conversion into a finite state machine to allow the validation of the specification. Figure 4 provides an overview of our approach, which consists of 3 main components: (1) Deep Graph generation (2) Translation of the Deep Graph into an SMV model (3) Verification of the SMV model.

### A. Deep Graph generation

To build the Deep Graph we start by initializing its entry point node $N_0$ using the CFG of the entry point

Listing 3: Method manager() in class Stack

```
1  public class Stack {
2    public void manager() {
3      push(data);
4      pop();
5      push(data);
6      peek();
7    }}
```

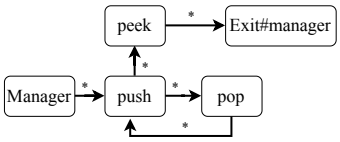


Fig. 6: DG of manager()

Listing 4: SMV model of the method Stack#manager()

```
1  MODULE main
2  VAR
3      eStack#manager : boolean;
4      r1 : boolean;
5      r2 : boolean;
6      state : {manager , ExitStack#manager , push , pop , peek,
           ERROR };
7  ASSIGN
8      init(r1) := FALSE;
9      init(r2) := FALSE;
10     init(eStack#manager) := TRUE;
11     init(state) :=manager;
12 next(state) := case
13     state = (manager) & eStack#manager : {push};
14     state = (push) & eStack#manager & r1: {pop};
15     state = (pop) & eStack#manager : {push};
16     state = (push) & eStack#manager & r2: {peek};
17     state = (peek) & eStack#manager : {ExitStack#manager};
18     state = (ExitStack#manager) & eStack#manager :
           {ExitStack#manager};
19     TRUE : {ERROR};
20 esac;
21 next(eStack#manager) := case
22     eStack#manager = TRUE : TRUE;
23     TRUE : FALSE;
24 esac;
25 next(r1) :=case
26     state =manager & r1 = FALSE : TRUE;
27     state = pop & r1 = TRUE : FALSE;
28     TRUE : FALSE;
29 esac;
30 next(r2) :=case
31     state = push & r1 = TRUE : FALSE;
32     TRUE : TRUE;
33 esac;
```

$V_e$ of the CG. Each node of the selected GFC is added to the DG. Then the CFG is traversed following the CFG's relation $F$ starting from $N_0$. Each edge in the DG is labelled using the name of the class and the method making the call. This is repeated for each method in the CG's set $V$, i.e., all methods of the system. For intraprocedural verification, this is enough. However, interprocedural verification requires an additional step.

At this stage, our DG contains an unconnected forest of graphs (the individual CFGs) with their edges labelled according to each graph's calling method. Interprocedural analysis requires us to connect these graphs. To do this, after each CFG has been added to the DG and the edges are labelled as described above, we select the entry point in the system and follow the relation $F$. We then check whether each method $k$ is a method of the client code. If this is the case, we add (a) an edge linking $k$ to its CFG and (b) another edge linking the corresponding CFG's exit point $M_f$ to $k$'s next method, $next(k)$.

For example, consider the CG in Figure 2, that shows that the method SafeClose() calls the client code method safeRelease(). The CFGs of the two methods are shown in Figure 5. As safeRelease() is a client code method, it may contain API calls. The algorithm therefore connects the node for SafeRelease() in the CFG of SafeClose() with its CFG, to create the DG shown in Figure 3. On the other hand, the node for Iterator.hasNext() is not a call to a client code method and is therefore not further expanded. Instead, it appears as a regular node in the DG.

We detail the algorithm for generating a Deep Graph in Algorithm 1. In it, we use the helper method prune(CFG) to remove unnecessary nodes. This procedure prunes instructions such as declarations and allows us to only keep nodes that contain method calls. In addition, for predicate nodes we add an identifier that allows us to distinguish them, especially when they are intertwined.

## B. Translating the Deep Graph into an SMV model.

Transforming the Deep Graph into a SMV model means translating each element of the DG into a different SMV model element, as shown in Table I. The rules of this translation are as follows:

- The DG becomes an SMV module;
- Elements of $N$ become elements of an enumerated variable State ;
- Each label in $\Delta$ becomes a boolean variable;
- $\Delta$ becomes rules of transition from one "State" to another;
- Each element of $\Delta : (s \xrightarrow{l} t)$ becomes a line in the transition rules using the following template: $State = (s) \wedge l \wedge rules : \{t\}$
- For each node with more than one output edge, we create determinism constraints that we call rules.

Consider the toy example of a program that uses a Stack abstract data type API as shown in Listing 3. The execution order of the method in Stack#manager is: push(),pop(),push(), and finally peek(). Its DG is shown in Figure 6. Starting from the entry node manager, we first encounter a node push, for the instruction on line 3 of Listing 3, followed by a node pop, for the instruction on line 4. We then return to the node push, as the command pop on line 3 is followed by a second instruction push in line 5. Consequently, there is a node for the peek instruction in line 6, and the exit node. This simple example illustrates how the DG is a lossy abstraction, as it can be traversed in more ways than the code in Listing 3 (see Table II). In this case, it is possible to recover some precision by adding additional rules to the SMV model to prevent this from happening. We show the complete SMV listing of the toy example in Listing 4, where these additional rules are encoded by the variables $r1$ and $r2$.

TABLE I: How DG concepts are represented in SMV

| DG concept | SMV concept |
|---|---|
| $N$ | Enumerated variable State |
| $N_0$ | init(State) |
| $\Delta$ | Next(State) |
| $n_{DG}$ | Each label is a boolean variable used in Next(State) rule |

TABLE II: Some execution traces of the DG in Figure 6.

| Trace | Call order |
|---|---|
| $t_0$ | push,pop,push,peek |
| $t_1$ | push,peek |
| $t_2$ | push,pop,push,pop,... |

### C. Model checking

As mentioned earlier, we assume that Tupac has at its disposal temporal API usage patterns, expressed as LTL properties that have been recovered by one of the techniques in Section III. The properties represent proper API usage. We do not make any assumptions about their quality in terms of accuracy and/or usefulness. Determining which patterns are of good quality is outside the scope of this paper.

We implemented Tupac using the symbolic model verifier NuXmv [4]. Given a property, we use NuXmv to verify that the SMV encoding of the structure of the code respects the pattern. As the encoding contains all possible traces of program execution (and potentially others), we can expect two types of returns from NuXmv: "TRUE" or a counterexample. NuXmv returns "TRUE" if it could not find any counterexample to the property. In other words, the pattern is respected, and the code does not contain API misuse. This is communicated to the user. NuXmv returns a counterexample, which corresponds to a path on the DG where the given property was violated. Such a path can be a true positive, i.e., a trace that violates the pattern and thus constitutes an example of API misuse. It can also be a false positive if (a) the pattern is not relevant in the given code fragment, or (b) the violation is due to the imprecision of the DG abstraction. To help the developer decide the best course of action, Tupac produces a visualization of the trace using GraphViz, which is presented to the user in a panel in Eclipse.

## VI. Preliminary Evaluation

The API usage patterns we are interested in define boundaries for the interactions between client applications and a given API. These interactions are generally captured by dynamic analysis, but this is not doable during the code writing phase. The goal of this preliminary evaluation is to assess to which extent our static-analysis-based approach Tupac allows to detect API usage pattern violations without the need to have a complete executable program. We target specifically the following three research questions:

- RQ1: To what extent Tupac is able to detect pattern violations using static analysis?
- RQ2: What are the benefits of interprocedural analysis compared to intraprocedural analysis?
- RQ3: Is Tupac fast enough to output usable results to the developer without stalling the coding workflow?

### A. Experimental Setup

To answer the research questions, we selected four open source programs that use four commonly used APIs, as shown in Table III. We then extracted temporal patterns to conduct our experiments as follows: We started by selecting a set of typical use cases for each program. We ran the corresponding use cases of each program and collected the execution traces. Then, for each program–API pair, we extracted from the traces the sequences of program-to-API calls. We used the sequences to mine LTL patterns for each API using the approach proposed by Saied et al. [23]. We thus obtained for each API as many pattern lists as programs using the API. To ensure that the mined patterns are valid, we kept only those that appear in all the programs' lists. This resulted in a list of 26 common patterns for the four APIs. We used these 26 patterns to answer the two first RQs. The patterns we considered are of similar complexity and size as in our previous work [23].

For RQ3, we used a larger sample of patterns without taking into account the overlapping between the programs' lists. We did this because for RQ3 we are only concerned with performance and not the validity of the patterns per se. For each API, we selected patterns with various complexities from the initial set of mined patterns. To sample different degrees of pattern complexity, we represented LTL patterns as syntactic trees and used the depth of the trees as a measure of complexity. The rationale behind this decision is that deeper pattern trees contain more logical propositions, requiring more time to check for violations. After the sampling, we obtained 124 patterns as shown in Table IV. Note that the number of patterns per API in the sample depends on the number of mined patterns and the distribution of depth among the patterns. For example, for util#Set, only 12 patterns were mined initially, and were thus all retained for RQ3. Moreover, the range of depth also varies from one API to another. Whereas in util#Set the pattern depths range from 2 to 14, the lower bound for io#File is 5 (no simple patterns) and the upper bound for the two remaining API is 7-8.

In order to better understand the contribution of each step in our analysis, we distinguish the time needed for the generation of the deep graph from the time to model check the patterns. We do this because the efficience of model checking is dependent on the model checking algorithm used, as well as the complexity of the patterns. Both of these factors are external to our approach, which can therefore benefit from new improvements to pattern mining techniques and model checking technology.

TABLE III: Dataset used

| Projects | Variable | util#Map | util#List | util#Set | io#File |
|---|---|---|---|---|---|
| Html compressor | $p_1$ | x | x | - | x |
| doc-to-pdf-converter | $p_2$ | - | - | - | x |
| jar2Java | $p_3$ | x | - | x | x |
| JTar | $p_4$ | x | - | x | x |

TABLE IV: Average syntax tree depth per single pattern

| API | #pattern | Depth |
|---|---|---|
| io#File | 72 | 5 to 11 |
| util#List | 14 | 2 to 7 |
| util#Set | 12 | 2 to 14 |
| util#Map | 26 | 2 to 8 |

## B. Results

RQ1: For this question, we assume that Tupac is deployed using the fully interprocedural approach. We want to assess the capacity of Tupac to find pattern violations, in a coding setting. To do so, we compared the output of Tupac with the ground truth obtained by manual inspection of the code and the execution traces. We manually analyzed each program to check how many patterns among the 26 are used correctly (measured by the metric CGT in Table V) and how many are violated or absent (metric VGT). Then, we used Tupac to check the 26 patterns on the four programs. The metric VTP in Table V indicates the number of patterns for which Tupac correctly found a counter example (true positives), and the metric VFP reports on the numbers of patterns incorrectly spotted by Tupac as violated/absent (false positives). Finally, we evaluated the precision, recall and F-measure as follows:
$Precision = VTP/(VTP + VFP)$
$Recall = VTP/VGT$
$F - score = 2 * (Precision * Recall)/(Precision + Recall)$
Given the results recorded in the interprocedural analysis section of Table V, we make the following observations.

First, we note that we have consistent results for the four programs. We had an almost perfect recall as Tupac was able to find almost all the violated or not used patterns except for one in the $p_2$.

Secondly, the precision obtained ranges from 0.53 to 1, which is higher than a random guess (0.5 for a binary decision). This level of precision is satisfactory considering the fact that static analysis, even sophisticated, cannot capture the sequences of interactions at the same level of precision as dynamic analysis. This fact explains most of the false positive we found.

In conclusion, Tupac is able to detect most of the pattern violations with a satisfactory precision, and in a context where a program cannot be executed.

RQ2: In order to perform an intraprocedural analysis, we checked all the methods that contain at least one API method call. Then we checked for the pattern violations using only the methods' call graphs. The results are presented in Table V. Although the recall is more or less the same for both analyses, the interprocedural alternative increased the precision for three out of the four programs by 10% to 27%. The benefits of the interprocedural analysis are also demonstrated by the fact that for $p_1$ and $p_4$, the precision of intraprocedural analysis is even lower than a random guess.

In conclusion, interprocedural analysis increases the precision of pattern violation detection as compared to intraprocedural analysis. This confirms the idea that API usage patterns are generally not limited to single methods but span the whole program.

RQ3: As mentioned earlier, we distinguish between the time required to build the deep graph and the time to model check the patterns.

The results concerning the construction of the DG are shown in in Table VI. We note that it took on average less than a second to create the DG for all the programs.

The results concerning the time for checking the patterns, are shown in Table VII. We note that it takes on average $695.84ms$ to check a pattern (between an average of $486.35ms$ and $814.99ms$). The high value for io#File can be explained by the high number of complex patterns. Another observation is that util#List and util#Set have similar times despite the fact that the latter has more complex patterns. This can be explained by the fact that some APIs are more used than others, causing them to be instantiated more frequently.

We conclude that, taking into account both the time for building the deep graph and for checking patterns, Tupac can be used in a coding context, without stalling the development. We draw our conclusion based on the average time for checking a single pattern. As we conjecture that APIs have a limited number of usage patterns, we believe this to be a reasonable approximation.

## C. Threats to Validity.

The evaluation presented in this paper is preliminary. Although we attempted to recreate realistic condition, many threats can limit its validity. First, the programs in this study were selected from the community website programcreek.com, which provides developers with code snippets containing API methods. These code snippets can be found in programs available on public repositories. The website allows developers to upvote the code snippets that helped them understand the use of a given API or otherwise downvote it. To select the APIs, we took the top four Java libraries in their "Top Java ranking" Classes. Next, we selected the programs that have the most positive votes while prioritizing projects that use APIs simultaneously.

TABLE V: Evaluation of Tupac accuracy for interprocedural and intraprocedural analysis.

| | | Interprocedural analysis | | | | Intraprocedural analysis | | | |
|---|---|---|---|---|---|---|---|---|---|
| Metric | Description | p1 | p2 | p3 | p4 | p1 | p2 | p3 | p4 |
| CGT | Ground Truth Correct | 17 | 1 | 15 | 16 | 17 | 1 | 15 | 16 |
| VGT | Ground Truth Violated/absent | 9 | 25 | 11 | 10 | 9 | 25 | 11 | 10 |
| VTP | True Positive Violated/absent | 9 | 24 | 11 | 10 | 9 | 25 | 11 | 10 |
| VFP | False Positive Violated/absent | 6 | 0 | 7 | 9 | 10 | 0 | 9 | 16 |
| Precision | | 0.6 | 1 | 0.61 | 0.53 | 0.47 | 1 | 0.55 | 0.38 |
| Recall | | 1 | 0.96 | 1 | 1 | 1 | 1 | 1 | 1 |
| F-score | | 0.75 | 0.98 | 0.76 | 0.69 | 0.64 | 1 | 0.71 | 0.56 |

TABLE VI: Average DG creation time per project in ms

| Project | Size(LOC) | DG(ms) |
|---|---|---|
| p1 | 5309 | 592 |
| p2 | 592 | 183 |
| p3 | 2879 | 758 |
| p4 | 1314 | 270 |

TABLE VII: Average verification time per single pattern

| API | Average(ms) | Av. per single pattern(ms) |
|---|---|---|
| io#File | 58679,50 | 814,99 |
| util#List | 10783,00 | 770,21 |
| util#Set | 8541,50 | 711,79 |
| util#Map | 12645,00 | 486,35 |

Regarding the programs used, we could have selected more and larger programs for a more comprehensive evaluation. We limited ourselves to these small/average size programs due to the effort needed to establish a ground truth by manually inspecting them to check for all possible applications of all the patterns.

We used execution traces to select patterns with which to experiment. The final 26 inferred patterns have a confidence between 0.9 and 1, according to the Texada LTL mining tool [11]. This means that for some of them, there are cases where the code does not fully satisfy the pattern. We believe that this simulates real situations well. In the future we intend to further experiment with extracting the patterns of an API from its documentation and randomly mutating the programs using the API to generate random violations.

Another possible threat to the validity concerns the choice of the typical use cases to produce the execution traces from which we mined the patterns. In order to mitigate this threat, we used examples provided in the documentation when available or the provided test suites.

## VII. Conclusion

We proposed Tupac, an approach that uses static interprocedural analysis to check the conformance of client code to temporal API patterns. The key idea of Tupac is that developers should be able to get feedback about correct API usage directly in the IDE without stalling their coding rhythm, even if their code is not complete and without needing execution. To verify the conformity of client code to API patterns, we start by creating models of the code: the Call Graph and Control Flow Graph. We use them to generate the Deep Graph (DG), which we convert to an SMV model. We then verify that this model satisfies the temporal API usage pattern, expressed as an LTL property using an off-the-shelf model checker. The results of the model checking are presented to the user directly in the IDE. This protects developers' productivity from context changes.

We conducted a preliminary evaluation of our approach on four Java projects and four APIs. We investigated whether Tupac can provide useful insights in a reasonable time frame. We also compared its effectiveness with a simpler, intraprocedural approach that does not depend on the construction of a Deep Graph. We found that Tupac can produce useful feedback in under 1 sec, which leads us to conclude that it can indeed be used without interrupting the regular flow of programming, e.g., by executing each time the developer saves the code.

While the results are encouraging, there is room for improvement, as our approach has some important limitations. For instance, as discussed in Section IV, we assume that the code does not contain any chained instructions. This is not always a reasonable assumption and preprocessing code to remove chains might result in additional complexity and imprecision.

Crucially, as shown in our preliminary evaluation, Tupac tends to produce many false positives. As it is based on static analysis, it does not handle dynamic aspects of Java, such as dynamic dispatch. Further, the DG is a abstraction of the set of possible execution paths and thus it encodes more behaviours than those present in the code. In its current version, our algorithm does not distinguish between the different objects a call is made. We have also not incorporated vacuity testing. Addressing these limitations by, for example, creating more sophisticated algorithms for generating more complete and sound DGs, is future work.

We also intend to do a deeper comparison between Tupac and related approaches. The main challenge is establishing a dataset to be used as a common comparison benchmark. Additionally, we want to further evaluate the practicality and usefulness of Tupac, by experimenting with more projects and APIs. A crucial challenge is establishing a comparison baseline, preferably without the need for manual analysis. To overcome this we are planing to use genetic programming to inject API misuses and simulate typical developer API usage.

## References


[1] F. E. Allen. Control Flow Analysis. In Proc. of a Symposium on Compiler Optimization, 1970.

[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. Investigating next steps in static api-misuse detection. MSR '19.

[3] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static API-misuse detectors. IEEE Transactions on Software Engineering, 45(12):1170–1188, 2018.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Proc. of Computer Aided Verification '02.

[5] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In Proc. of OOPSLA'08.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In Proc. of ICSE '99.

[7] H. J. Goldsby and B. H. C. Cheng. Automatically Discovering Properties That Specify the Latent Behavior of UML Models . In Proc. of MODELS '10.

[8] M. Huth and M. Ryan. Logic in Computer Science: Modelling and reasoning about systems. 2004.

[9] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. 2006.

[10] S. A. Kripke. Semantical considerations on modal logic. Acta Philosophica Fennica, 1963.

[11] C. Lemieux, D. Park, and I. Beschastnikh. General LTL specification mining (t). In In Proc. of ASE'15, pages 81–92. IEEE, 2015.

[12] D. Lo, S. Khoo, and C. Liu. Mining temporal rules for software maintenance. J. Softw. Maintenance Res. Pract. '08.

[13] G. C. Murphy. Beyond integrated development environments: Adding context to software development. In Proc. of ICSE-NIER '19.

[14] D. Nam, A. Horvath, A. Macvean, B. Myers, and B. Vasilescu. Marble: Mining for boilerplate code to identify api usability problems. 2019.

[15] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Grapacc: A graph-based pattern-oriented, context-sensitive code completion tool. In Proc. of ICSE '12.

[16] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta. Focus: A recommender system for mining api function calls and usage patterns. In Proc. of ICSE '19.

[17] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J.r M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In Proc. of SIGSOFT '09.

[18] E. Raelijohn, M. Famelis, and H. Sahraoui. A vision for helping developers use apis by leveraging temporal patterns. In FormaliSE '19.

[19] S. Rastkar, G. C. Murphy, and A. W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In ICSM '11.

[20] M. Robillard and R.t Deline. A field study of api learning obstacles. Empirical Software Engineering '11.

[21] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong. On-demand developer documentation. In Proc. of ICSME '17.

[22] M. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo. Improving reusability of software libraries through usage pattern mining. Journal of Systems and Software 2018.

[23] M. Saied, E. Raelijohn, E. Batot, M. Famelis, and H. Sahraoui. Towards assisting developers in api usage by automated recovery of complex temporal patterns. Information and Software Technology 2020.

[24] A. Shatnawi, H. Shatnawi, M. Saied, Z. A. Shara, H. Sahraoui, and A. Seriai. Identifying software components from object-oriented apis based on dynamic analysis. In Proc. of ICPC '18.

[25] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In Proc. of ASE'11, 18(3):263–292, 2011.

[26] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In In Proc. of ESEC/FSE'07, pages 35–44, 2007.

[27] M. Wen, Y. Liu, R. Wu, X. Xie, S. Cheung, and Z. Su. Exposing library api misuses via mutation analysis. In ICSE '19.

[28] C. Xu, Xi. Sun, B. Li, Xi. Lu, and H. Guo. Mulapi: Improving api method recommendation with api usage location. J. Syst. Softw. '18.

[29] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In Proc. of ICSE '06.

[30] H. Zhong, T. Xie, Lu Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In Proc. of ECOOP '09.