Checking temporal patterns of API usage without code execution

Erick Raelijohn, Michalis Famelis, Houari Sahraoui

FormaliSE 2021
IDEs are Awesome!

- Automatic code generation
- Organize imports
- “On the fly” compilation
But not as much as they could be

Adele

Android API

API Knowledge

• Read and understand the documentation
• Look for examples
• Seek community support

Productivity!

Code

Feedback

Feedback from other sources (e.g., execution, bug reports)

IDE
Our Vision

- Read and understand the documentation
- Look for examples
- Seek community support

Productivity!

Safer Usage

API Knowledge

Code

Feedback

IDE

API Knowledge

Adele
Temporal Usage PAttern Checker (Tupac) for APIs

Many existing automated API property inference techniques

<<Existing work>>
Extract
Usage pattern

<<Our contribution>>
Integrate
Temporal pattern

Challenge/contribution:
Check temporal patterns with static analysis on incomplete programs
Outline

• Motivation

• Approach

• Preliminary Evaluation

• Conclusion
Basic Assumptions

• Object Oriented (OO) paradigm

• Check pattern properties *in the IDE* as part of regular coding rhythm

• Be able to check patterns on **incomplete code**

• **Static analysis**: no need to run the code, no reliance on tests, traces

• Temporal API usage properties as **external inputs**
  (from documentation, or an extraction tool)

• Properties language: Linear Temporal Logic - **LTL**
Tupac’s Approach

Code in IDE

Temporal API usage properties (LTL)

“Deep Graph”
Core Concepts

- **Control Flow Graph (CFG) of **method**
  Can be used for intraprocedural analysis

- **Call Graph (CG) of OO system**
  Shows intraprocedural dependencies

- **Deep Graph (DG) of OO system**
  Combines the CG and all CFGs of a system
  Each control transition annotated with the method call triggered it

- "Trace": a path on the DG
  Executing the code *might* produce this sequence of API calls
  I.e., the DG may contain more behaviours than the code
Deep Graph Construction

Inputs: a set of control flow graphs, a call graph
Output: a deep graph

1. **Prune and slice** inputs to only keep API calls and calls to other system methods
2. Following the CG, **annotate** each edge of CFGs with the caller method
3. **Connect the forest** of annotated CFGs based on the CG
public class DataManager extends SQLiteClosable {

    public void safeClose() {
        Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
        this.safeRelease();
        while (iter.hasNext()) {
            Map.Entry<SQLiteClosable, Object> entry = iter.next();
            SQLiteClosable program = entry.getKey();
            if (program != null)
                program.onAllReferencesReleasedFromContainer();
        } 
        onAllReferencesReleased();
    }

    public void safeRelease() {
        for (SQLiteCompiledSql compiledSql : this.CQueries.values())
            compiledSql.releaseSqlStatement();
        CQueries.clear();
    }
}
Intuition: Follow the calls

Call Graph

Control Flow Graph
Result: Deep Graph
Model Checking the Deep Graph

• DG translated to a NuSMV module
  • State: method and API calls
  • Transition relation: caller annotations + constraints to enforce determinism

• API property patterns in LTL
  • Assumed as given externally

• Possible results:

<table>
<thead>
<tr>
<th></th>
<th>TRUE</th>
<th>FALSE + counterexample</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The pattern is respected</td>
<td>DG path where the pattern is violated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- True positive: API misuse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- False positive: (a) pattern irrelevant for code fragment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (b) imprecision of DG</td>
</tr>
</tbody>
</table>

Tupac produces a visualization to help make sense
Outline

• Motivation

• Approach

• Preliminary Evaluation

• Conclusion
Research Questions

🎯 RQ1: How good is Tupac for detecting pattern violations?

🌐 RQ2: Interprocedural vs intraprocedural analysis

🏃‍♂️ RQ3: Is Tupac fast enough be usable without stalling devs’ workflow?
Setup

• 4 open source programs
  HtmlCompressor, doc-to-pdf-converter, jar2Java, JTar

• 4 commonly used APIs

• Typical use cases → Trace collection → Sequences of API calls

• API pattern mining using an existing technique [1]

  • RQ1, RQ2: only keep 26 shared patterns

  • For RQ3: sampled 124 patterns of various AST complexities


Independent from Tupac! Any extraction technique might be used
## RQ1 Results

1. Results consistent for p1-p4
2. Great recall
3. Precision better than random (>50%)
RQ2 Results

Intraprocedural analysis: within a single method; not following method calls
Might it be a cost-effective shortcut to the expensive DG construction?

Using only the Control Flow Graph

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Interprocedural analysis</th>
<th>Intraprocedural analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGT</td>
<td>Ground Truth Correct</td>
<td>p1: 17, p2: 1, p3: 15, p4: 16</td>
<td>p1: 17, p2: 1, p3: 15, p4: 16</td>
</tr>
</tbody>
</table>

Precision = \( \frac{VTP}{VTP + VFP} \), Recall = \( \frac{VTP}{VGT} \), F-score = \( \frac{2 \times (\text{Precision} \times \text{Recall})}{\text{Precision} + \text{Recall}} \)

1. Similar recall
2. Bad precision (worse than random)

\( \geq \)
RQ3 Results

1. On average, less than 1 second to create DG
   (*DGs are reusable!*)

2. On average, 0.7 seconds to check a single pattern
Outline

• Motivation

• Approach

• Preliminary Evaluation

• Conclusion
Temporal Usage PAttern Checker (**Tupac**) for APIs

Many existing automated API property inference techniques

<<Existing work>>

Extract

Usage pattern

<<Our contribution>>

Integrate

Temporal pattern

IDE

Recommendation

API
Tupac’s Approach

Code in IDE

Temporal API usage properties (LTL)

“Deep Graph”
Preliminary Evaluation Findings

Tupac is reasonably good at detecting pattern violations

Interprocedural is better than intraprocedural analysis

Tupac is fast enough to be usable without stalling devs’ workflow
Limitations and Future Work

• Does not handle complex code structures (e.g., chained instructions)

• Many false positives
  • No vacuity testing: what patterns are even relevant?
  • DG encodes more behaviours than the code
  • Do not distinguish between different objects making the call

• Need more thorough evaluation, validation with users, and deeper comparison with similar approaches
  • Challenge: no common static analysis benchmark
Checking temporal patterns of API usage without code execution

Our Vision

Temporal Usage PAnttern Checker (Tupac) for APIs

Tupac’s Approach

Findings

How good is Tupac for detecting pattern violations?

Interprocedural vs intraprocedural analysis

Is Tupac fast enough be usable without stalling devs’ workflow?