Toward a Critic System for API Client Code using Symbolic Execution

Daqing Hou, Chandan R. Rupakheti
Department of Electrical and Computer Engineering
Clarkson University, Potsdam, New York 13699
{*dhou, rupakhcr*}*@clarkson.edu*

***Abstract*—It is well-known that API's can be hard to learn and use. Although search tools can help find related code examples, API novices still face other significant challenges such as evaluating the relevance of the search results. To help address this as well as the more general and broader problem of information needs in using API's, we propose a critic system that offers *explanation*, *criticism*, and *recommendation* for API client code. Our system takes as input API use rules from API experts, performs symbolic execution to check that the client code has followed these rules properly, and generates advices and recommendations as output to help improve the client code. To assess the practicality of developing and implementing API use rules for the proposed critic, we conducted an initial empirical study on the actual problems that the Java Swing users had while programming GUI layouts. The proposed critic system was able to solve many of the API use problems.**

***Keywords*-API; Critic; Symbolic Execution**

I. Introduction

Behind each API (Application Programming Interfaces) is a system that offers proven solutions for a set of common problems in some domain. The API facilitates the access to such a system so that new systems can be built on top of it. To be effective in using the API, one must learn enough of the domain, its problems and solutions, and how to map between them appropriately. For a system of rich functionalities with a large problem and solution space, learning its API can be a substantial endeavor [5], [6], [9].

Due to time pressure and an urge to solve problems quickly, many programmers prefer to learn API's on demand and learn by doing. That is, they try to learn just enough of an API so that they can solve the current task. Search tools can partially support this practice by helping locate relevant code examples, for example, [1], [4]. But programmers, especially novices, cannot always formulate good queries for what they are looking for. Furthermore, even if they find relevant code examples, they would still face a significant challenge to understand and evaluate them for relevance. Programmers can also seek help from online forums, but there can be a significant time lag before they can get one.

Figure 1 depicts the current state of practice for API-based programming and the role of a proposed critic system in the overall API usage scenario. Given a problem to be solved, initially an API novice may have only a vague idea for the prospective solutions. To develop a concrete plan to solve the problem with the API, he needs to consult extensively

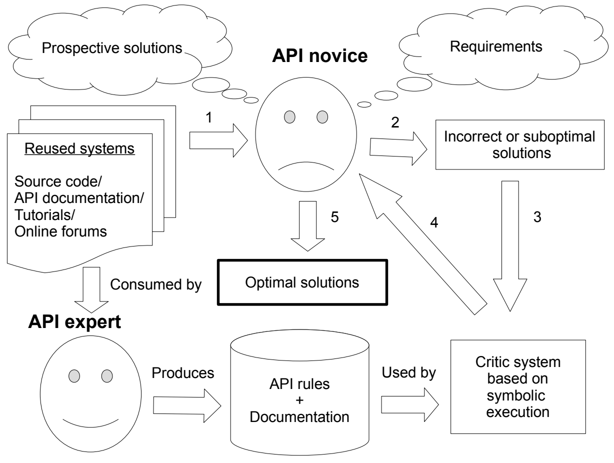


Figure 1. The state of practice for API-based programming and the role of the proposed critic system.

such artifacts as API documentation, tutorials, forums, and code samples. When the novice starts writing code, with only limited knowledge of the API, his or her solution is often incorrect or suboptimal.

It would be ideal to engage a human expert for help, but experts are scarce. As an alternative, we are investigating a critic system [2], [10] that can advise the novice online while the code is being written. More specifically, it can

1) *explain* the interactions of multiple API elements,
2) *criticize* the improper use of the API, and
3) *recommend* other relevant API elements for future use.

The system symbolically executes API client code [7]. Based on the result of the symbolic execution as well as its knowledge about API use rules, the system generates feedback for the programmer's code.

As shown in Figure 1, the proposed critic system needs a set of API use rules and the associated documentation that explains these rules. The API rules would be prepared by a human expert who has substantive experience with the API. With these, our system will be able to provide contextual advice about the programmer's code. With multiple interleaved rounds of coding and critiquing, the system would incrementally direct the programmer toward a correct and optimal solution. We believe that such a system has a potential to bridge the long-standing information gap

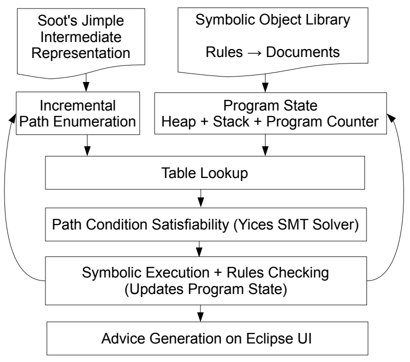

Soot's Jimple Intermediate Representation
Symbolic Object Library
Rules → Documents
Incremental Path Enumeration
Program State
Heap + Stack + Program Counter
Table Lookup
Path Condition Satisfiability (Yices SMT Solver)
Symbolic Execution + Rules Checking
(Updates Program State)
Advice Generation on Eclipse UI

Figure 2. Architecture of the proposed critic system.

between the API designers and the application programmers.

II. Architectural Overview for the Critic

Figure 2 depicts the overall architecture for the proposed critic system. The technical foundation for our system is symbolic execution [7], which requires an entry method to start its execution. Our current prototype supports the Java AWT/Swing API, particularly, the layout of GUI components. For this API, the entry methods are those that instantiate a top-level window such as a `JFrame` and a `JDialog`, or any subclass of these.

The control-flow graph of an entry method is traversed in depth-first order to enumerate paths and perform symbolic execution. Non-library methods are inlined to produce interprocedural paths. Loops and recursive calls are expanded up to a specified bound.

The program state at any control point consists of the heap and the stack, which contain variables and their symbolic values, as well as the program counter, which marks the current statement on a feasible path. The path conditions at the entry to each branch are represented as predicates over the symbolic program state. These predicates are conjoined and passed to a constraint solver (we plan to use the Yices SMT Solver) for checking satisfiability and pruning infeasible paths. To execute each statement on a feasible path, the symbolic values that the statement refers to are first looked up from the stack and heap. The statement is then executed against these values, according to its semantics.

When backtracking from one branch to another, the program state must be restored to the point right before the immediate common predecessor of the two branches. We achieve this by maintaining the history for each variable in the program state at the granularity of branches. However, for efficiency, within a branch, assignments overwrite previous values that variables may have.

The state of a program can be checked against the API use rules at several points of interest. The points of interest could be before or after an API method, or an event handling method, is called. We plan to make the points of interest configurable so that other possible points can be defined.

The listeners registered on GUI widgets are also monitored. After a top-level symbolic widget is made visible in the main control flow, a combination of different GUI events of a pre-specified length are generated and executed to check the effect of the GUI events on the program state. This technique has also been used by others in the generation of GUI test-cases [3], [8].

The core of our critic system consists of a set of base classes. By extending these classes, the semantics of a new API can be modeled soundly and conservatively as Java objects and methods. These symbolic objects collectively form the symbolic object library in Figure 2. Thus, supporting a new API amounts to parameterizing our critic system with a symbolic object library. Currently, the checking of API use rules and the presentation of related documentation are directly implemented in Java together with symbolic objects.

III. A Motivating Example

In this section, we illustrate the three forms of advice (*explanation*, *recommendation*, and *criticism*) that our critic can produce. As an example, consider the following message copied from the Swing Forum [1]:

> *Please help me, how to use the Grid layout.*
> *In my code i have to use Grid layout*
> *I have to print the 4 labels in one row. and 4 textfields in the next row [to make] a table.*

Listing 1 shows the code mentioned in the above message (for presentation, simplified to contain only two labels and two textfields). Figure 3 shows the GUI that the current code produces as well as the desired GUI. Table I depicts the program states at important control points after the code is executed symbolically. Program states are checked against API use rules to infer the current status of the program and the programmer's intents and goals, and to offer advice.

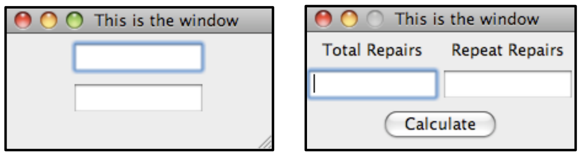
A. Explanation

The facts holding in program states can be used to help the programmer understand why the program exhibits a certain behavior. For instance, from line 9 in Table I, the critic detects that the layout manager of the content pane of `frame` is set to `BorderLayout`. But the default layout manager for the content pane of a `JFrame` is already `BorderLayout` (line 8). Therefore, line 9 is redundant. Such behavior can be explained to the user. Although IDEs also explain individual API elements by showing Javadoc comments, they do not explain the interaction of multiple API elements. Note that a criticism and a recommendation may also contain explanations.

B. Criticism

Criticisms are produced when the client code violates the state invariants of the API objects. For example, to

[1]http://forums.oracle.com/forums/thread.jspa?messageID=5737802 (All URLs verified Nov 30, 2011)



(a) Current GUI (b) Desired GUI

Figure 3. Current GUI produced by Listing 1 and the desired GUI.

be visible, a non-top-level GUI widget must participate in a GUI hierarchy rooted at a top-level window. The critic detects that this is not the case for `labelPanel`, `buttonPanel`, and `example`, because their parents are null at line 23 of Table I. Note that although Eclipse would also warn user about the unused `example` object (line 26), it does not warn about `labelPanel` and `buttonPanel`.

Listing 1. A program for producing a table-like GUI (modified).

```
1  class XYZ extends JPanel {
2   JLabel lblTotalRepairs = new JLabel(``Total Repairs'');
3   JLabel lblRepeatRepairs = new JLabel(``Repeat Repairs'');
4   JTextField totalRepairs = new JTextField(8);
5   JTextField repeatRepairs = new JTextField(8);
6   JButton button1 =new JButton(``Calculate'');
7   XYZ() {
8    JFrame frame = new JFrame(``This is the window'');
9    frame.getContentPane().setLayout(new BorderLayout());
10
11   JPanel labelPanel =new JPanel();
12   JPanel fieldPanel =new JPanel();
13   JPanel buttonPanel =new JPanel();
14
15   setLayout(new GridLayout(2,2));
16
17   labelPanel.add(lblTotalRepairs);
18   labelPanel.add(lblRepeatRepairs);
19   fieldPanel.add(totalRepairs);
20   fieldPanel.add(repeatRepairs);
21   buttonPanel.add(button1);
22   frame.getContentPane().add(fieldPanel, BorderLayout.CENTER);
23   frame.setVisible(true);
24  }
25  public static void main(String[] args) {
26   XYZ example = new XYZ();
27  }
28 }
```

C. *Recommendation*

By inferring a programmer's intents from program states and anticipating his or her needs, our critic can recommend both alternative solutions and additional API elements that may be needed next. Assume that as a solution for the problem mentioned in Section III-B, the programmer has added the two orphan panels to the frame. Now, at line 23, where the GUI is made visible, the critic detects that `labelPanel` contains two `JLabel`'s as its children, and `fieldPanel` contains two `JTextField`'s (from facts at line 21 of Table I). By examining this GUI data structure resulting from symbolic execution, our prototype critic is able to infer that the programmer is attempting to create a 2-by-2 table. The critic is also able to conclude that this way of making a table is problematic as it will be impossible to properly align a label and its corresponding text field. Instead, such a table view can be made in a single container using the `SpringLayout`, `GridLayout`, or `GridBagLayout`. All of this information has been added to the critic as a rule of recommendation. Thus, the critic would recommend this to the programmer.

Table I
SIMPLIFIED SYMBOLIC PROGRAM STATES FOR LISTING 1.

Line #	Facts that hold after line # for symbolic objects

7	example.parent = null
	example.layout = FlowLayout()
	example.lblTotalRepairs = JLabel(...)
	...
	example.button1 = JButton(...)

8	frame.title = "This is the window"
	frame.visible = false
	frame.contentPane.children = []
	frame.contentPane.layout = BorderLayout()
	frame.contentPane.layout.properties = []

9	frame.contentPane.layout = BorderLayout()
15	example.layout = GridLayout(2,2)

21	labelPanel.children = [lblTotalRepairs, lblRepeatRepairs]
	fieldPanel.children = [totalRepairs, repeatRepairs]
	buttonPanel.children = [button1]

22	frame.contentPane.children = [fieldPanel]
	fieldPanel.parent = frame.contentPane
	frame.contentPane.layout.properties = [CENTER:fieldPanel]

23	frame.visible = true
	labelPanel.parent = null
	buttonPanel.parent = null
	example.parent = null

IV. PRELIMINARY FINDINGS FROM EMPIRICAL STUDY

To get a sense of the actual problems that API users may face and to gain initial experience with developing and implementing API rules for the proposed critic, we analyzed the API discussions in *Swing Forum* [2]. To narrow down the scope of our analysis, we chose to focus on issues related to GUI layouts. To expedite the process of finding relevant posts, we searched the forum with *layout* as keyword. This returned 274 discussion threads.

We sequentially analyzed 124 of these 274 threads. Typically, a thread may contain multiple posts with questions, answers, and code snippets. A code snippet is usually a self-contained, compilable program that is provided so that other forum participants can run it to diagnose the problems. To understand the discussion of each thread and identify and fix the problems ourselves, we carefully read the text and ran the code snippets contained in each thread.

61 of the 124 threads were deemed unclear, too general, or lacked sample code for us to concretely assess whether

[2]http://forums.oracle.com/forums/forum.jspa?forumID=950

our critic could be helpful. We concluded that 63 threads, which contain 145 code snippets, could be helped by our proposed critic. From these 63 threads, we identified and implemented a total of 41 API rules in our first prototype (20 for criticism, and 21 for recommendation). Since these 41 rules helped 145 code snippets, on average, each rule was applied 3.5 times. In future work, we will look into generalizing the rules to make them more powerful and hit more cases. To further illustrate the scope of possible advice, we sample some of the identified API rules from this study.

A. Additional Sample Criticism Rules from the Study

The criticism rule about unused widgets presented in Section III-B involves state invariants. There are other common mistakes that although may be correct behaviorally, are not best practices at best, or reflect a lack of understanding on the part of the programmer. As an example, consider the following code [3]:

```
a=new JPanel();
b=new JPanel();
c=new JPanel();
c.setLayout(new BorderLayout());
c.add(a);
c.add(b);
```

This code calls the `add` method on a `JPanel` that uses `BorderLayout` with a single argument, which will add the component to the `CENTER` location of the panel. Since both `a` and `b` are added to the center, the net effect is that only `b` is managed by the `BorderLayout`; at runtime, only `b` can be visible. But the programmer in the post wants `a` and `b` to be both visible and positioned vertically. This can be achieved by calling:

```
c.add(a, BorderLayout.NORTH);
c.add(b, BorderLayout.SOUTH);
```

The critic would explain this behavior to the programmer and recommend the above as a potential solution.

B. Additional Sample Recommendation Rules from the Study

The rules for recommendation advise the programmer about alternative solutions as well as additional API elements that may be needed next. The example presented in Section III-C provides an alternative solution for a problem. In addition, additional API elements can be recommended based on their relevance to API elements that are currently used in the code. The notion of relevance appears to be a key factor for defining this kind of recommendations.

One way that an API element becomes relevant to the other API elements in the existing code is when they are all parts of the same solution for a certain problem. For instance, almost every time when a `JFrame` object is created, `pack` is called to lay out the frame, and `setVisible(true)` to make it visible.

[3]http://forums.oracle.com/forums/thread.jspa?messageID=5701289

A group of API elements may also be considered "relevant" to each other because it is easy to confuse them. Although these API elements share some similarity either by their names or purposes, they are meant to be used in different contexts for different purposes. Thus, it can be useful to offer clarification for them as a group. For example, there are several API methods available for a `JLabel` that are related to alignment and horizontal positioning: `setAlignmentX()`, `setHorizontalAlignment()`, and `setHorizontalTextPosition()`. They look superficially similar but actually serve three different purposes. This is not obvious just by reading their Javadoc. The critic can recommend additional clarification documentation.

V. Conclusion

We describe our initial experience with an API critic that is intended to significantly improve the experience of learning to use a new API. The foundation for our critic consists of program states, obtained from symbolic execution, and API use rules. We have implemented a first prototype for the Java Swing API. To gain an understanding of the nature of API use rules, and the practicality of developing them, we have analyzed a set of Swing newsgroup discussions in an initial empirical study. This initial experience indicates that it is feasible to develop API rules to capture recurring API use problems. However, to make the critic cost-effective, with an acceptable ratio of return-on-investment, future work should be directed to creating API use rules that can capture a sufficiently large number of instances of API use problems.

References

[1] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *CHI*, 2010, pp. 513–522.

[2] G. Fischer, A. C. Lemke, T. Mastaglio, and A. I. Morch, "Using critics to empower users," in *CHI*, 1990, pp. 337–347.

[3] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Test generation for graphical user interfaces based on symbolic execution," in *AST*, 2008, pp. 33–40.

[4] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *ICSE*, 2010, pp. 475–484.

[5] D. Hou and L. Li, "Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions," in *ICPC*, 2011, pp. 91–100.

[6] D. Hou, C. Rupakheti, and H. Hoover, "Documenting and evaluating scattered concerns for framework usability: A case study," in *APSEC*, 2008, pp. 213 –220.

[7] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[8] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *WCRE*, 2003, pp. 260–269.

[9] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE Softw.*, vol. 26, pp. 26–34, 2009.

[10] C. R. Rupakheti and D. Hou, "Satisfying programmers' information needs in API-based programming," in *ICPC*, 2011, pp. 250–253.