CLARKSON UNIVERSITY

A Critic for API Client Code using Symbolic Execution

A Thesis by

Chandan R. Rupakheti

Department of Electrical and Computer Engineering

Submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Electrical and Computer Engineering

May 2012

Accepted by the Graduate School

Date DEAN

The undersigned have examined the thesis entitled **A Critic for API Client Code using Symbolic Execution** presented by Chandan R. Rupakheti, a candidate for the degree of Doctor of Philosophy, Electrical and Computer Engineering and hereby certify that it is worthy of acceptance.

_____

Date

EXAMINING COMMITTEE

_____

Dr. Susan Conry

_____

Dr. Christopher Lynch

_____

Dr. Jeanna Matthews

_____

Dr. Robert Meyer

_____

ADVISOR Dr. Daqing Hou

Abstract

It is well-known that APIs can be hard to learn and use. To cope with difficulties in using APIs, programmers browse the Internet for code samples, tutorials, and API documentation. In general, it is time-consuming to find relevant help from the plethora of information on the web. While search tools may help programmers find code snippets, novices often have difficulty in formulating a useful search query and in assessing the quality and relevancy of the search results. To help address the broad problems of *finding*, *understanding*, and *debugging* API-based solutions, this thesis presents a new kind of critic system called **CriticAL** (A **C**ritic for **A**PIs and **L**ibraries) that offers *recommendations*, *explanations*, and *criticisms* for API client code.

CriticAL takes API usage rules as input, performs symbolic execution to check that the client code has followed these rules properly, and generates advice as output to help improve the client code. While several past studies have focused on answering why frameworks are hard to learn and use, very few have provided systematic data that can be used directly in building such a tool. We conduct a manual case study of 150 discussion threads from the Java Swing forum and derive a set of framework rules that can be directly used in API tooling efforts of CriticAL. We show that there are recurring patterns in problems the programmers face while using the Swing framework that can be supported through CriticAL. We were able to capture the nature of the recurring patterns precisely as API usage rules. We demonstrate the usefulness of CriticAL by applying it to real-world examples derived from the Java Swing Forum and through a formative user case study.

Acknowledgements

I am indebted to the support and guidance of my advisor, Dr. Daqing Hou. He taught me how to conduct research. He showed me how to write research articles. He guided me, inspired me, and brought the best in me. I thank him from the bottom of my heart. Special thanks go to Dr. Christopher Lynch, Dr. Christino Tamon, Dr. Jeanna Matthews, Dr. Susan Conry, and Dr. Robert Meyer for their guidance and constructive comments on the thesis and for their help with my academic career.

I want to dedicate this thesis to my loving grandmother Annapurna Rupakheti and to the loving memory of my grandfather Giri Raj Rupakheti who would be proud and happy to see the first Ph.D. in his family. I am blessed to have a wonderful family who taught me to work hard under every circumstance. I thank my mom (Chanda Sharma), my dad (Chandrika Prasad Rupakheti), and my brother (Chetan Raj Rupakheti) for their love and support. I am fortunate to have whom I consider the most beautiful woman in the world as my fiancée. She has endured me and motivated me through all of the ups and downs in my life in the last 8 years. Thank you Manila for your love, affection, and friendship.

Finally, thank you all of my friends at Software Engineering Research Laboratory (Patricia Deshane, Ferosh Jacob, Cheng Wang, Yuejiao Wang, Xiaojia Yao, Lin Li, Dave Pletcher, Lingfeng Mo, Siyuan Bao, Tiantian Zhao, and Ying Zhang) for sharing your lives with me in the lab and making school a fun-filled learning experience.

List of Publications

Publications that contributed directly or indirectly to this thesis:

1. C. R. Rupakheti, D. Hou, *Evaluating Forum Discussions to Inform the Design of an API Critic*, In Proceedings of IEEE International Conference on Program Comprehension (ICPC), June 2012, 10 pp. (to appear).

2. C. R. Rupakheti, D. Hou, *CriticAL: A Critic for APIs and Libraries*, In Proceedings of IEEE ICPC, June 2012, 3 pp. (to appear).

3. C. R. Rupakheti, D. Hou, *Finding Errors from Reverse-Engineered Equality Models using a Constraint Solver*, April 2012, 10 pp., (submitted for peer review).

4. C. R. Rupakheti, D. Hou, *EQ: Checking the Implementation of Equality in Java*, In Proceedings of IEEE International Conference on Software Maintenance (ICSM), September 2011, pp. 590-593.

5. C. R. Rupakheti, D. Hou, *Satisfying Programmers' Information Needs in API-Based Programming*, In Proceedings of IEEE ICPC, June 2011, pp. 250-253.

6. C. R. Rupakheti, D. Hou, *An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java*, In Proceedings of IEEE Working Conference on Reverse Engineering (WCRE), October 2010, pp. 205-214.

7. C. R. Rupakheti, D. Hou, *An Empirical Study of Design and Implementation of Object Equality in Java*, In Proceedings of Center for Advanced Studies on Collaborative Research Conference (CASCON), ACM, October 2008, pp. 111-125.

8. D. Hou, C. R. Rupakheti, H. J. Hoover, *Documenting and Evaluating Scattered Concerns for Framework Usability: A Case Study*, In Proceedings of IEEE Asia-Pacific Software Engineering Conference (APSEC), December 2008, pp. 213-220.

Contents

List of Algorithms

List of Tables

Listings

List of Figures

Chapter 1

Introduction

Behind each Application Programming Interface (API) 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 [33, 34, 48].

Past studies on API usage have elicited three fundamental challenges in reuse based development [17, 31, 33, 34, 43, 49]:

1. *finding* the right API elements for the programming task,
2. *understading* their use to achieve the desired goal, and
3. *debugging* the client code that uses them.

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. While search-based tools partially help solve the first problem, novices still face a significant challenge in using such tools especially when they lack the knowledge of the framework's design to formulate the right search queries [33, 41].

In general, a reuse-oriented system may not be used in the best possible ways that it

is originally designed for. Instead, users often settle with a suboptimal set of available solutions that are just enough for their current tasks [17]. To address this problem, Fischer envisions an architectural design for a development environment that encompasses such tools as visualization, explanation, recommendation, and critics, to help programmers work with the framework and APIs [17].

The kind of environments that Fischer proposes are aimed at helping bridge the gap between the *situation model* and the *system model*. The situation model is a programmer's mental model about the task or problem needed to be solved, and it is often imprecise and informal. The system model, on the other hand, is about the actual running system. The system model is precise and formal, and can be expressed by such technical artifacts as source code that a machine can understand and execute, and design documentation.

In translating a problem in the situation model to a solution in the system model, a programmer may face several challenges. Given a problem, the programmer may not know whether a solution is possible and what process he needs to follow to create the solution. When there are multiple possible solutions, he may not know which one suits his situation the best. Finally, a programmer may make mistakes in executing a complex solution procedure.

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 or have limited time. To help, this thesis presents a critic system [1] [20, 53–55] that can advise the novice online while the code is being written in his development environment. 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.

[1] The **CriticAL** project (A **Critic** for **A**PIs and **L**ibraries) can be found at `http://sf.net/p/critical`. All URLs verified on May 3, 2012.

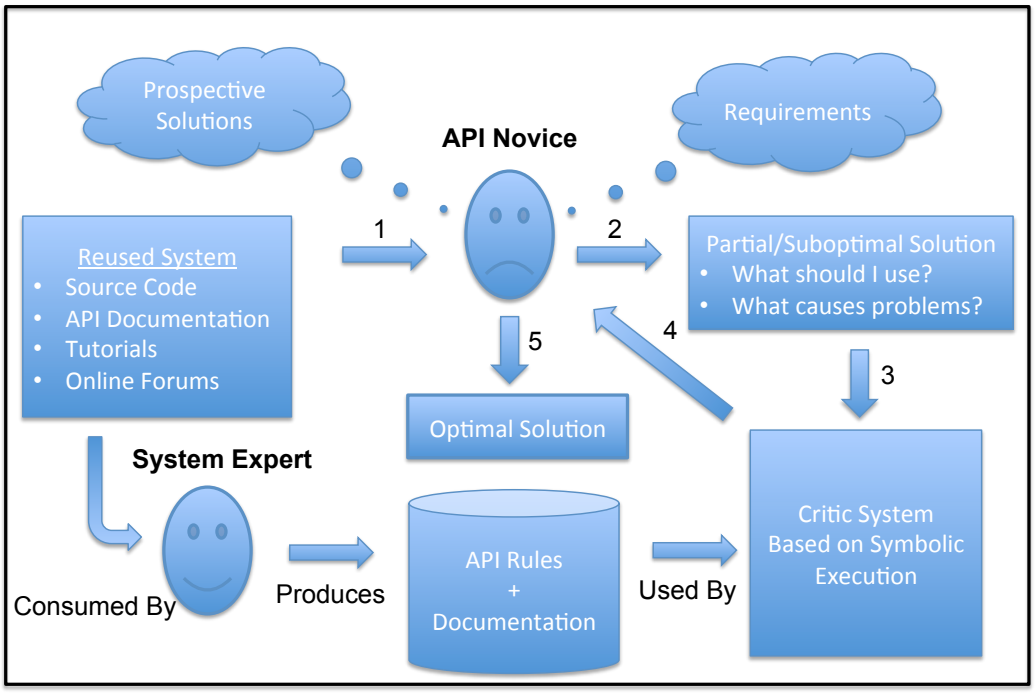
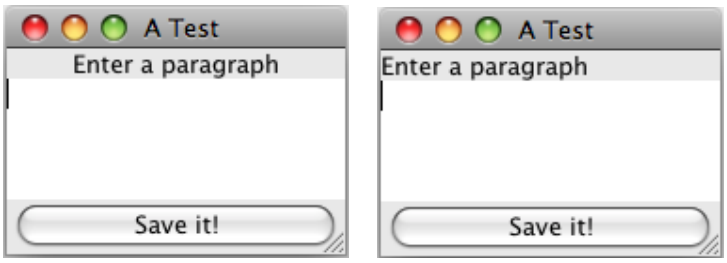


Figure 1.1: Role of our critic system in API-based programming.

Figure 1.1 depicts the current state of practice for API-based programming and the role of a critic system in the overall API usage scenario. In the literature, a computer program that critiques human-generated solutions is called *a critic* [59]. Critics have been successfully applied in clinical medicine management, engineering design, word processing, and software engineering. Our API critic is expected to help bridge the long-standing information gap between API designers and application programmers, and thereby increase the quality of the novice's code, as well as move him or her toward being an expert.

As shown in Figure 1.1, our critic system requires a set of API usage rules and associated documentation to explain these rules. The API rules specify special program states for which advice should be produced. A human expert who has substantive experience with the API is responsible for producing the rules. The system symbolically executes API client code [40]. Based on the resulting symbolic program states as well as its knowledge of API usage rules, the system generates contextual advice for the programmer's code. It is expected that with multiple interleaved rounds of coding and critiquing, the system would incrementally



(a) Desired design. (b) Alignment problem.

Figure 1.2: User requirement and achieved solution.

direct the programmer toward a correct and optimal solution. Hence, such a system has a potential to bridge the long-standing information gap between the API designers and the application programmers.

1.1 Motivating Examples

In this section, we will present a high-level overview of CriticAL with the help of two motivating examples. The first example will show the application of CriticAL in reuse-based development using the Java Swing API [2]. The second example will illustrate the symbolic execution mechanism used by CriticAL also using the Swing API.

1.1.1 Case 1: Application of CriticAL in Reuse-based Development

Learning to Use the API

Consider a novice programmer wanting to learn the Java Swing API and undertaking the problem of designing the Graphical User Interface (GUI) shown in Figure 1.2(a). Let us assume that he does not know much besides that there is a `JFrame` class that he could use to make the window. This is a sound assumption as there are Swing forum users with such problems, e.g., [3] [4]. Figure 1.3(a) shows that the user manages to create an empty `JFrame` object in the Eclipse IDE and probably with the help of its content assist, sets the title of

[2] `http://docs.oracle.com/javase/tutorial/uiswing/`
[3] `https://forums.oracle.com/forums/thread.jspa?messageID=5768843`
[4] `https://forums.oracle.com/forums/thread.jspa?messageID=5848584`

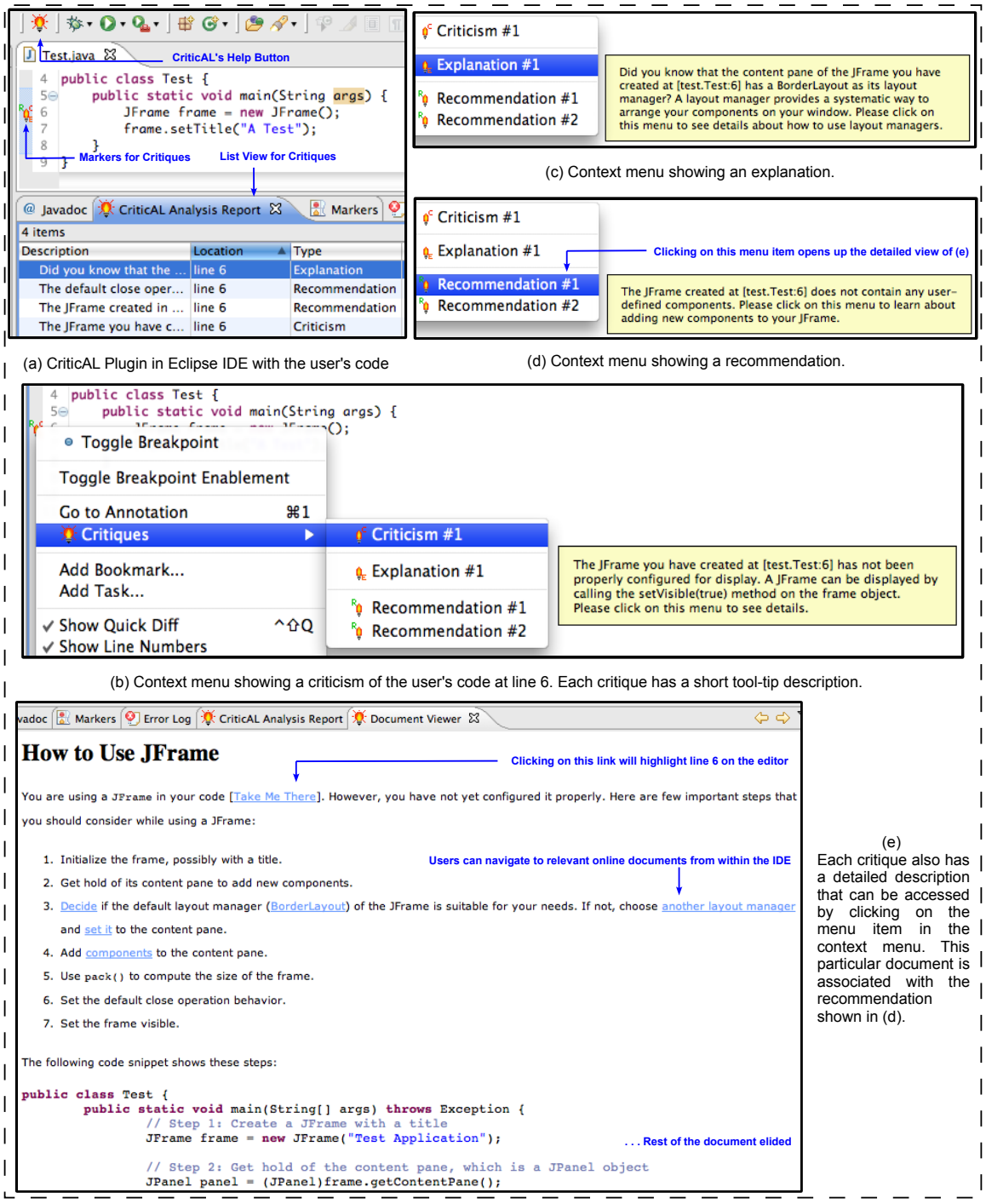


Figure 1.3: CriticAL helping a programmer build the Swing application shown in Figure 1.2(a) using a `JFrame`.

the window and then gets stuck as he does not know what to do next. Starting from such a preliminary stage, CriticAL will come to his aid by guiding him through the coding process, where it will read his code and provide critiques to help him achieve the desired solution iteratively within the IDE.

The user presses the help button of CriticAL (Figure 1.3(a)) after getting stuck. Equipped with the rules and documentation for the Swing framework, CriticAL symbolically executes the user's code and finds that the `JFrame` object is missing some of the important properties in the execution path, which triggers some of the critiquing rules. Figure 1.3(a) shows markers for those rules in line 6.

The user right-clicks over the marker to get the context menu as shown in Figure 1.3(b). The context menu lists all of the criticisms, explanations, and recommendations made by CriticAL for the given line number. On hovering over each of the menu items under *Critiques*, CriticAL displays a short tool-tip message describing the critique. In this case, Figure 1.3(b) shows a tool-tip for the criticism generated due to the violation of the rule that *a top-level GUI widget if initialized, must eventually be visible.* Figure 1.3(c) shows the explanation generated when *the default layout manager of the content pane of the* `JFrame` *is used instead of the user-configured one.* Figure 1.3(d) shows the recommendation generated when *the content pane of a* `JFrame` *is empty.* When the user clicks on a menu item, CriticAL opens a detailed view showing the description of the problem with code snippets as well as links to the related online documents. In this case, Figure 1.3(e) shows the detailed document for the recommendation of Figure 1.3(d). Hence, the user can now learn about `JFrame` and `BorderLayout` and even copy-paste the code snippet on the editor.

Using the API

Let's assume that the user now understands how to use `JFrame` and `BorderLayout` and produces the code as shown in Listing 1.1. Nevertheless, when he runs the code, he gets the window shown in Figure 1.2(b) where the label is aligned at the left. If he uses Eclipse's content assist (Figure 1.4), he finds that the `setAlignmentY(float)` method

Listing 1.1: The user's code resulting in the `JFrame` of Figure 1.2(b).

```
1 // Initialize a frame and other components
2     JFrame frame = new JFrame(“A Test”);
3     JPanel panel = (JPanel)frame.getContentPane();
4     JLabel lbl = new JLabel(“Enter a paragraph”);
5     JTextArea textArea = new JTextArea(4,15);
6     JButton button = new JButton(“Save it!”);
7 // Add components to the content pane
8     panel.add(lbl, BorderLayout.PAGE_START);
9     panel.add(textArea, BorderLayout.CENTER);
10    panel.add(button, BorderLayout.PAGE_END);
11 // Compute the frame's size and display
12    frame.pack();
13    frame.setVisible(true);
```

could be used to align components horizontally but to no avail. He further gets confused when he discovers that there are two more methods with similar sounding functionalities viz. `setHorizontalAlignment()` and `setHorizontalTextPosition()` and none of their Javadocs communicate a proper usage scenario. CrticAL would again help him by recommending a document that explains their behavior and the situation where each of them suit better as follows:

- `setAlignmentX()` and `setAlignmentY()` of `JComponent` used with parameters such as `TOP_ALIGNMENT` and `CENTER_ALIGNMENT`. They are used to position widgets in a cell of a `BoxLayout`.

- `setHorizontalAlignment()` and `setVerticalAlignment()` of `JLabel` and `JButton` with parameters such as `LEFT` and `CENTER`. They are used to align the content of a widget within itself.

- `setHorizontalTextPosition()` and `setVerticalTextPosition()` of `JLabel` and `JButton` with parameters such as `LEFT` and `TOP`. They are used to align the text within a `JLabel` and a `JButton` relative to their icon image.

After reading this document, the user would insert `lbl.setHorizontalAlignment(JLabel.CENTER)` in line 7 to finally get the desired GUI of Figure 1.2(a). Note that this iterative

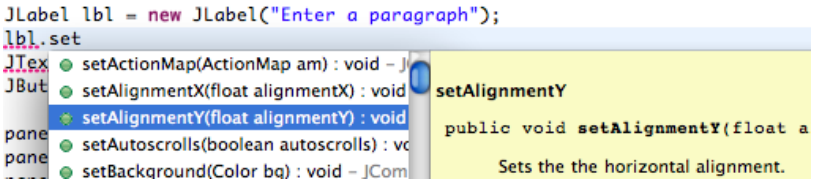


Figure 1.4: Eclipse's content assist showing alignment related methods.

process of providing contextual recommendations is a rather ambitious goal and we do not claim that we have completely achieved it. We recommend bits and pieces that each may serve a particular moment in a long programming process. To achieve this ambitious goal, the tool needs to adapt to the different level of expertise and provide sufficient coverage. Nevertheless, these recommendations are useful. The basis for these recommendations are the normal way of using the API and occurrence of similar problems in the Swing forum, which we will discuss in Chapter 2. The cost of recommending in these cases is minimal because these recommendations passively appear as a marker in the IDE and do not interfere with the user's regular programming, but the benefit can be substantial.

1.1.2 Case 2: Gentle Introduction to the Use of Symbolic State

To illustrate the three forms of advice (*explanation*, *recommendation*, and *criticism*) that our critic can produce through symbolic execution, consider this message copied from the Swing Forum [5]:

> *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.*

Figure 1.5(a) shows the code mentioned in the above message (for presentation, simplified to contain only two labels and two textfields). Figure 1.6 shows the GUI that the current code produces as well as the desired GUI. Using CriticAL for help, the programmer may press the CriticAL button as shown in Figure 1.5(a). As a result, CriticAL symbolically executes the code to create the states of the program, which are shown in Table 1.1.

[5] http://forums.oracle.com/forums/thread.jspa?messageID=5737802

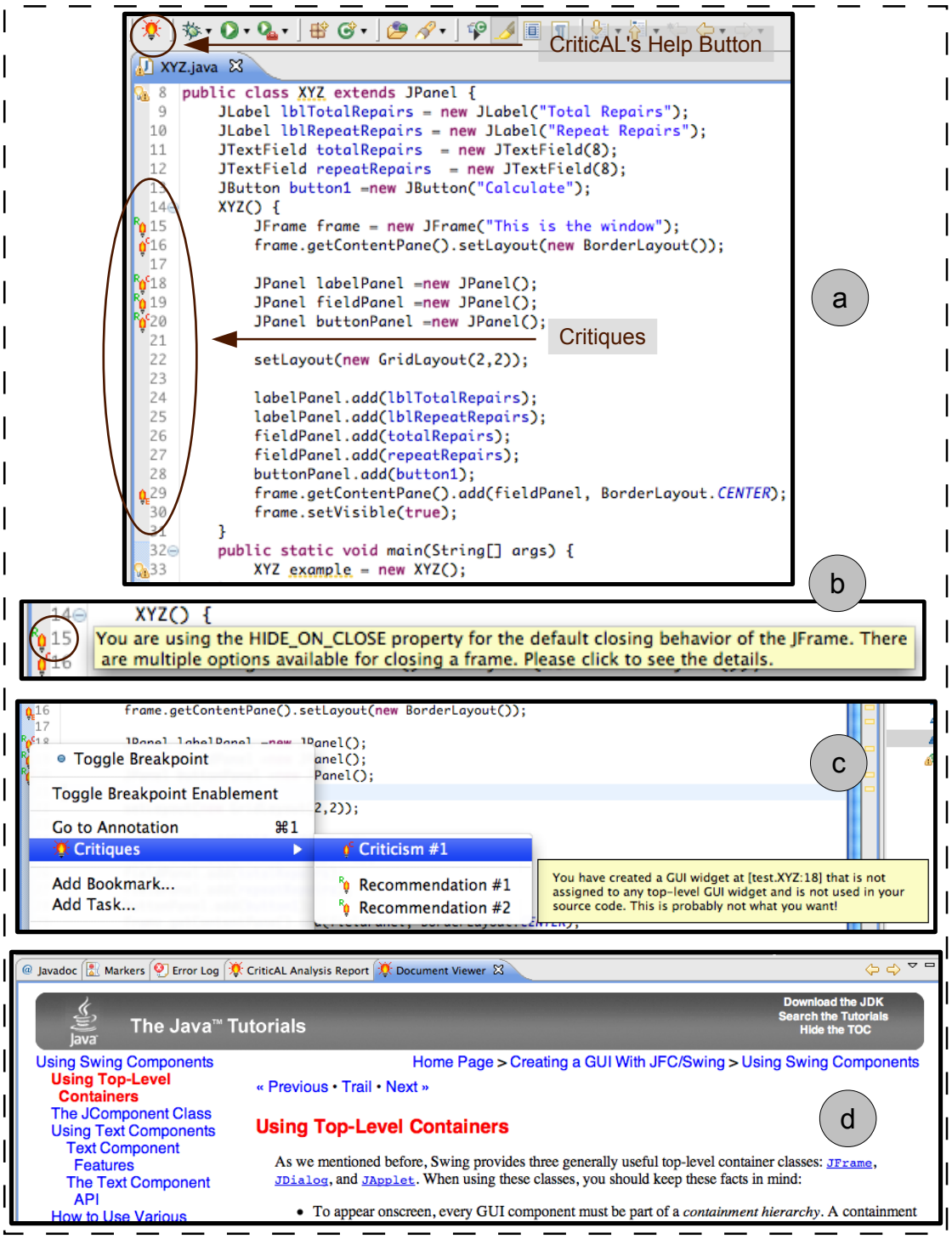


Figure 1.5: CriticAL helping a programmer build the Swing application of Figure 1.6(b).

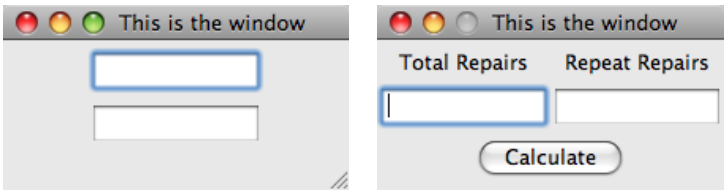
Table 1.1: Symbolic states for the program of Figure 1.5(a).

Line #	**Facts that hold after line # for symbolic objects**
14	example.parent = null example.layout = FlowLayout() example.lblTotalRepairs = JLabel(...) ... example.button1 = JButton(...)
15	frame.title = "This is the window" frame.visible = false frame.contentPane.children = [] frame.contentPane.layout = BorderLayout() frame.contentPane.layout.properties = []
16	frame.contentPane.layout = BorderLayout()
22	example.layout = GridLayout(2,2)
28	labelPanel.children = [lblTotalRepairs, lblRepeatRepairs] fieldPanel.children = [totalRepairs, repeatRepairs] buttonPanel.children = [button1]
29	frame.contentPane.children = [fieldPanel] fieldPanel.parent = frame.contentPane frame.contentPane.layout.properties = [CENTER:fieldPanel]
30	frame.visible = true frame.defaultCloseOperation = HIDE_ON_CLOSE labelPanel.parent = null buttonPanel.parent = null example.parent = null

Our critic checks the program states against API use rules to infer the current status of the program as well as the programmer's intents and goals, and to offer advice. Generated advice is presented as markers on the ruler of the text editor (the left-hand side of Figure 1.5(a)), indicating that CriticAL has critiques (E: Explanation, R: Recommendation, and C: Criticism) for the code at the corresponding lines.

Explanation

The facts holding in program states can be used to help the programmer understand why the program exhibits a certain behavior. For instance, CriticAL finds that a component is added to the center location of the `BorderLayout` that manages the content pane (line



(a) Current GUI (b) Desired GUI

Figure 1.6: Current GUI produced by Figure 1.5(a) and the desired GUI.

29, Table 1.1). On hovering over the explanation marker (line 29, Figure 1.5(a)), CriticAL presents a tool-tip description explaining that the added component will grow with the window as the window is resized. Although IDE's 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.

Recommendation

By inferring a programmer's intent 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. For instance, by default a `JFrame` has the `HIDE_ON_CLOSE` property set as the default close operation (line 30, Table 1.1). When the user presses the close button of the frame, this property will only hide the frame without actually disposing the frame object. The recommendation at line 15 (Figure 1.5(b)) presents the user with other available options for the closing behavior, e.g. `EXIT_ON_CLOSE`. The three other recommendations at lines 18, 19, and 20 inform the user how to control the horizontal/vertical gaps and the alignments for the `JPanel`s through their `FlowLayout`s.

Criticism

Criticisms are produced when the client code violates the pre-/post-conditions and the state invariants of the API objects. For example, to be visible, a non-top-level GUI widget must participate in a GUI hierarchy rooted at a top-level window. Our critic detects that this is not the case for `labelPanel` (created at line 18) and `buttonPanel` (line 20) because

their parents are null at line 30 of Table 1.1. Since there is more than one critique (R and C) at line 18, CriticAL offers a context menu for accessing these critiques (Figure 1.5(c)). On clicking the menu item in the figure, CriticAL presents a detailed document related to the problem, as shown in Figure 1.5(d). The document from the Java Swing tutorial is used in this case. In general, every critique has a short tool-tip description and a detailed explanation document, which can be stored either locally or remotely on the Internet.

Assume that to fix the problem, the programmer added the two orphan panels to the frame. Now, at line 30, 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 28 of Table 1.1). By examining this symbolic GUI data structure, our critic infers that the programmer is creating 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 can be made in a single container using `SpringLayout`, `GridLayout`, or `GridBagLayout`. This information has been added to the critic as a rule of criticism.

1.2 Contributions

This dissertation tries to answer the following research questions:

1. What are the characteristics of API usage problems?
2. Can we formulate useful rules that capture the problems?
3. Do API usage problems recur such that we can justify the investment made on preparing the rules and documentation by API experts?
4. Can we model the behavior of an API abstractly and apply API rules on them to produce critiques? Is a critic system feasible for a relatively large API such as Swing?

The main contributions of this thesis are the answers to the four key research questions:

1. We present a conceptual classification framework for reasoning about the real-world problems programmers face while using APIs by manually analyzing 150 discussion threads in the Java Swing forum. We classify the problems based on how they could be helped by a critic.

2. We show that API problems recur in practice and API-usage rules could be developed to address them.

3. Developing conceptual rules is not enough. We present a symbolic-execution-based, extensible framework for supporting the API-based programming practice. Symbolic execution tools have been widely used for program verification. To the best of our knowledge, CriticAL is the first application of symbolic execution in actively helping programmers learn and use APIs. We show that most of the API usage rules developed in the study of the Java Swing forum could be easily supported by CriticAL.

4. We evaluate CriticAL by conducting a formative user case study and by applying it to the code collected from the Java swing forum and the official Swing tutorials.

1.3 Overview of the Thesis

In this chapter, we introduced CriticAL, a symbolic-execution-based static analysis framework for supporting API-based programming. To provide useful help, CriticAL must be pre-configured with rules and documentation. In Chapter 2, we will conduct a case study of the Java Swing Forum to illustrate the process of extracting such rules and documentation. Chapter 3 will discuss the design of CriticAL, the semantics of symbolic execution, and algorithms used for efficiently cloning symbolic objects and execution stacks to minimize memory footprints. We will discuss the implementation detail of extending CriticAL to support new APIs and libraries in Chapter 4. In Chapter 5, we will evaluate CriticAL based on the real programs collected from the Swing forum as well as through a formative study conducted in an undergraduate GUI development class at Clarkson University in Fall

2011. We will compare and contrast this work with other related work in Chapter 6 and finally, Chapter 7 concludes the thesis.

Chapter 2

Case Study

Learning to use a software framework and its API can be a major endeavor for novices. To help, we have built a critic to advise the use of an API based on the formal semantics of the API. Specifically, the critic offers advice when the symbolic state of the API client code triggers any API usage rules. To assess to what extent our critic can help solve practical API usage problems and what kinds of API usage rules can be formulated, we manually analyzed 150 discussion threads from the Java Swing forum. We categorize the discussion threads according to how they can be helped by the critic. We find that API problems of the same nature appear repeatedly in the forum, and that API problems of the same nature can be addressed by implementing a new API usage rule for the critic. We characterize the set of discovered API usage rules as a whole. Unlike past empirical studies that focus on answering why frameworks and APIs are hard to learn, ours is the first designed to produce systematic data that is directly used to build an API support tool.

2.1 Motivation

As mentioned in Chapter 1, we distinguish among three kinds of critiques: *criticisms* ("this code behavior is inappropriate"), *explanations* ("what has caused the code to behave this way"), and *recommendations* ("you may need this next"). They are broadly designed to address the respective well-known challenges in *debugging*, *understanding*, and *finding*

relevant solutions [43]. However, to justify, and more importantly, to guide the further development of our critic, we need solid empirical data about API use from the field. Although several studies have been directed toward answering the question why APIs are hard to learn and use [31, 33, 34, 43, 49], prior research has not produced concrete data that can be directly used to build an API support tool. This motivated us to conduct a case study to analyze and collect such data from the programming discussions in the Swing Forum [1]. Specifically, we want to answer two research questions:

- What are the characteristics of API usage problems? How are the needs for the three kinds of critiques grounded empirically?
- Do similar problems recur? To what extent can our critic help advise practical API usage problems?

The results of our study, in the form of a spreadsheet along with the source code collected, are available online at `https://sourceforge.net/projects/critical/files/results/`.

The rest of the chapter is organized as follows. Section 2.2 presents our research method. Results of our study are reported in Sections 2.3 (criticisms), 2.4 (explanations), and 2.5 (recommendations). We summarize our main findings in Section 2.6. Section 2.7 discusses the threats to validity of this study. Finally, Section 2.8 concludes the chapter.

2.2 Research Method

In this case study, we are interested in answering two research questions discussed in Section 2.1. To this end, we have conducted a case study of the online programming questions in the Java Swing Forum. To conduct an in-depth exploration, we have chosen to narrow down the scope of the analysis and focus our study on problems related to *GUI composition and layout* in the Java Swing API. GUI composition and layout is an essential topic in GUI programming that is backed by a strong design, but which many novices have great difficulty with. Therefore, lessons learned from this study are likely to be generalizable.

[1] `https://forums.oracle.com/forums/forum.jspa?forumID=950`

We employ Eisenhardt's methodology for case study research [15]. In our study, each forum discussion thread represents a real-world scenario (or a case, in terms of case study research) where somebody is having certain problems with the API. A typical discussion thread contains multiple posts with questions, answers, and code examples. Many posted code examples are self-contained, compilable programs that forum members can run to assess the problems. Since there were too many discussion threads in the Swing Forum to go through manually (more than 46,000 threads and more than 211,000 messages), to expedite the process, we searched the forum with the keyword *layout* [2]. This query returned 264 threads. Starting from the first thread, we classified each thread under the rules that apply. We stopped at 150 threads because we felt that we hit the point of diminishing return where we did not have to formulate new rules to solve the problems asked by the original posters (OPs).

Eisenhardt's method dictates that the observer must be intimately familiar with the cases/subjects. To ensure that, we have paid close attention to the fine details in the cases. Specifically, in addition to reading the text throughout, we compiled and ran each code, sometimes with necessary modifications, in order to explore each case in detail. This not only helped us get the full understanding of each case, but also resulted in a set of 90 test cases for testing our critic. During the process, we occasionally referred to the online tutorials [3] and the API reference manual for help.

As Eisenhardt describes, the process of identifying categories from case data and encoding them is highly iterative. In our study, the categories are the specific *API use rules* that can be used to trigger helpful advice based on the symbolic program states of the API client code. (For examples, see Section 2.3 Criticisms, Section 2.4 Explanations, and Section 2.5 Recommendations). We label each thread with all rules that we conclude are useful for the thread. To be more thorough, we took help from Dr. Hou in the coding and classification process. Disagreements between us, in terms of both the interpretation and the classifica-

[2]Search URL: `https://forums.oracle.com/forums/search.jspa?threadID=&q=layout&objID=f950&dateRange=all&userID=&numResults=15&rankBy=10001`

[3]`http://docs.oracle.com/javase/tutorial/uiswing/`

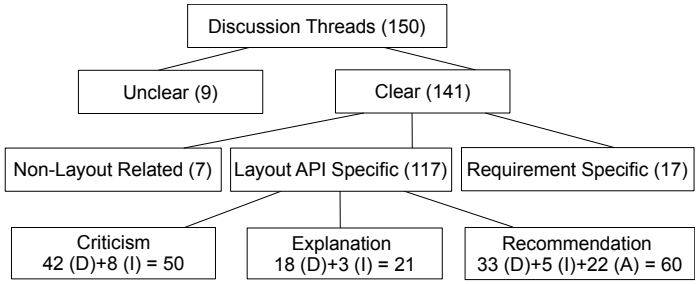


Figure 2.1: Classification of 150 Swing Forum discussion threads. **D**: directly helpful to an OP's core problems; **I**: indirectly helpful; **A**: most likely helpful. A thread helped by multiple critiques of the same kind is counted only once. Since a thread may be helped by more than one kind of criticism, recommendation, and explanation, the total number for critiques is more than 117.

tion of the discussion threads, were resolved through numerous discussions over the course of more than ten months. The results of our analysis, in the form of a spreadsheet along with the code collected from the forum, are available online.

Figure 2.1 shows the top-level categories that lead to a final categorization of the 150 threads according to how they can be supported by the three kinds of critiques:

- There were 9 threads for which we did not understand what was the OP's key question (Original Poster), due to either poor English or unclear presentation, e.g [4].

- Among the 141 clear cases, 7 were not related to the layout API that this study is focused on. For example, in one case [5], although the word "layout" appears in the messages, the OP was asking about how to change the layout of the keyboard from English to German.

- 17 of the 141 clear threads were about application specific requirements related to layout. For example, one OP posted a GUI design diagram and asked how to achieve it [6]. In such cases, there is not much that our critic can help other than consulting a human expert.

- The rest of the 117 threads were related to *layout and GUI composition* that we

[4] `https://forums.oracle.com/forums/thread.jspa?messageID=5838236`
[5] `https://forums.oracle.com/forums/thread.jspa?messageID=5833268`
[6] `https://forums.oracle.com/forums/thread.jspa?messageID=5888922`

conclude can be supported through the three forms of critiques, that is, *criticisms* (50 threads), *explanations* (21 threads), and *recommendations* (60 threads).

Our result shows that code is commonly used for communicating about API usage problems. Based on the analysis of the 141 clear threads, we found that 42.6% of the OPs (60 threads) posted code when asking about their problems, and that in 38.3%, or 54, of the 141 threads, some forum users replied with code. In this study, we have collected a total of 90 runnable Java programs.

We have found encouraging evidence that our critic can be a valuable complement for humans. This is because our critic has been found to be helpful not only for cases that have been provided a solution but also for those without a solution. In particular, we have found that 44%, or 62, of the 141 clear threads did not contain a solution. Interestingly, 75.8%, or 47, of the 62 threads can be supported by the critiquing rules that we developed in this study, 16 through criticisms, 10 through explanations, and 21 through recommendations. Why were these threads not answered by forum participants? This is most likely because many of them (25 threads) contain a long piece of code and parsing through such long code to find problems is a cumbersome task. Our critic can be particularly valuable for such cases, complementing human capabilities. In subsequent sections, we discuss in detail the critiquing rules that we have identified for criticisms, explanations, and recommendations.

2.3 Criticisms

A criticism informs the programmer about some undesirable behavior in the API client code. Table 2.1 depicts the list of specific criticism rules that we have identified in this study and how many times each rule has been found to be useful. Section 2.3.1 introduces these rules and briefly discusses how they are checked by our critic. Five cases are presented as examples in the remaining subsections.

2.3.1 API Criticism Rules

A GUI programmed with the Swing API is essentially a tree data structure. The root of the tree is a special top-level widget such as a `JFrame` or a `JDialog`, leaves are made of basic GUI widgets such as `JLabel`, `JTextField`, and `JButton`, and internal nodes a container such as `JPanel`, which contains other widgets recursively. To be displayed, a widget must have a location and a size computed or explicitly set. A container may rely on a `LayoutManager`, which is essentially an algorithm, to automatically compute the size and location for each of its child widgets, based on layout-specific constraints and strategies. Swing provides several built-in layout managers, each with a different layout strategy, such as `FlowLayout`, `BoxLayout`, `GridLayout`, `GridBagLayout`, and `SpringLayout`. Rather than using a layout manager, a programmer also has the option to manually specify the size and location for each widget, which is also known as *absolute positioning*.

The following criticism rules are found useful to enforce the internal consistencies of the GUI tree:

- *Orphan GUI Objects*: To be visible, all GUI objects must be part of a GUI tree rooted at a top-level component such as a `JFrame` or a `JDialog`.
- *Parent Switching*: When the containing GUI tree is invisible, moving a widget between two containers has no effect and, thus, should be avoided.
- *Missing Layout Constraints*: When using a `SpringLayout`, necessary constraints for the container as well as its widgets must be specified to get the desired effect.
- *Misplaced Layout Constraints*: Some layout managers, such as `BorderLayout` and `GridBagLayout`, require layout specific constraints to position the child components. When widgets are added to a container that uses a layout manager, only constraints specific to the layout manager should be used.
- *One Layout, One Container*: The relationship between layout managers and containers must be one-to-one. Each layout manager maintains the size and position

Listing 2.1: `JPanel`s sharing the same layout manager.

```
1 BorderLayout layout = new BorderLayout();
2 JPanel ui = new JPanel(layout);
3 JPanel preview = new JPanel(layout);
4 JPanel figures = new JPanel(layout);
```

information of the child widgets for a container. Sharing a layout manager may result in unpredictable GUI behavior. The example in Listing 2.1 shows three panels sharing the same `BorderLayout` [7].

- *Content Mismatch*: When a container is made visible, it must have the same set of child widgets as its layout manager.

- *Positioning and Sizing Constraints*: When a layout manager is used by a container, calling `setLocation()`, `setSize()`, and `setBounds()` methods on child components have no effect and should not be used. When `null` layout is used, the `setPreferredSize()`, `setMinimumSize()`, and `setMaximumSize()` methods have no effect and should not be used. The `JFrame.pack()` method should be used only when the content pane of the frame has a layout manager or when it has an explicitly set preferred size.

- *Dynamic GUIs*: When the content of a container is changed, it must be revalidated and repainted for the change to take effect.

Each API targets to solve a particular set of problems. APIs, thus, have some usage conventions. While it is not necessarily always wrong to use an API in a way deviating from conventions, such a use is nonetheless uncommon, often showing some confusion or neglect of the programmer. Spotting such deviations can thus be useful. We have identified two common deviations from conventions:

- *Components Resizing Behavior*: Not all components are meant to be resized in both directions. By convention, widgets such as `JButton` and `JLabel` should not be resized in either direction. Widgets such as `JTextField` and `JPasswordField` could grow

[7] `https://forums.oracle.com/forums/thread.jspa?messageID=5890601`

Table 2.1: List of helpful criticisms discovered in the forum (D: Direct, I: Indirect, T: Total). The Return on Investment (ROI) is calculated by dividing the total number of helped cases by the total number of rules.

API Criticism Rules	**D / I / T**
Postconditions	
Orphan GUI Objects	6 / 0 / 6
Missing Layout Constraints	2 / 2 / 4
Parent Switching	2 / 0 / 2
Misplaced Layout Constraints	3 / 1 / 4
Invariants	
Content Mismatch	4 / 0 / 4
Dynamic GUIs	4 / 0 / 4
One Layout, One Container	3 / 0 / 3
Preconditions	
JFrame.pack() Constraints	5 / 4 / 9
Positioning and Sizing Constraints	8 / 0 / 8
Deviation from Usage Conventions	
Components Resizing Behavior	10 / 4 / 14
Table Design	10 / 2 / 12
Total	**58 / 12 / 70**
Return on Investment	**6.36**

horizontally but not vertically. A violation of such conventions is often undesirable [8]. When they are violated, it can be useful to teach the users how to prevent the widget from stretching.

- *Table Design*: A table-like GUI design is conventionally achieved using one of `GridLayout`, `GridbagLayout`, and `SpringLayout` with a single container. Two adjacent containers cannot be used to create a table-like design. This is because it would be hard, if not impossible, to align the GUI widgets contained in the two containers.

As shown in Table 2.1, the criticism rules are enforced as preconditions, postconditions, and invariants for the GUI layout API. Since none of these rules looks overly complicated, it is probably safe to speculate that programmers encounter problems mainly due to lack of awareness of these simple rules. Hence a tool like our critic can be very useful.

[8] `https://forums.oracle.com/forums/thread.jspa?messageID=5854070`

Listing 2.2: Code for Case 1 (modified).

```
1 public class CoordinateLayout extends SpringLayout {
2  SpringLayout main; Container cont;
3  public CoordinateLayout(Container ct) {
4   main = new SpringLayout(); cont = ct; ...
5  }
6  public void addComponent(Component comp, int x, int y) {
7   main.putConstraint(WEST, comp, x, WEST, cont); // X−axis
8   main.putConstraint(NORTH, comp, y, NORTH, cont); // Y−axis
9  }}
10 public class CoordinateLayoutTest {
11  public static void main(String[] args) {
12   JFrame frame = new JFrame(“TEST”);
13   JPanel pane = new JPanel();
14   CoordinateLayout layout = new CoordinateLayout(pane);
15   pane.setLayout(layout);
16   JLabel aLabel = new JLabel(“First Name:”);
17   JButton aButton = new JButton(“First”);
18   // layout contains widgets but container does not
19   layout.addComponent(aLabel, 5, 5);
20   layout.addComponent(aButton, 15, 5);
21   frame.setSize(300,300);
22   frame.setContentPane(pane);
23   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24   frame.setVisible(true);
25  }}
```

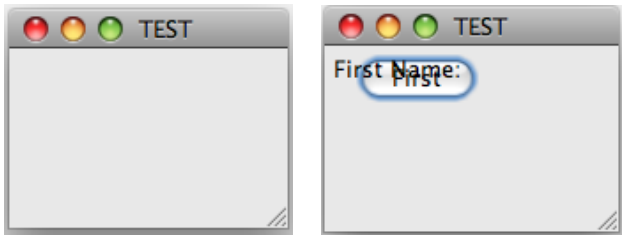
2.3.2 Case 1 (Orphan Objects, Content Mismatch, Missing Constraints)

The code for Case 1 is shown in Listing 2.2 [9], where the `CoordinateLayout` class extends Swing's layout manager `SpringLayout` to specify the position of a widget relative to its container. Our critic reveals three problems.

First, the `SpringLayout` object created at line 4 and referenced by `main` is not used by any container. Hence it becomes an orphan GUI object. In fact, the `main` object is unnecessary, and the `main.putConstraint()` at lines 7 and 8 should be replaced by calls to `this.putConstraint()`. Although our critic does not directly give this advice, pointing out the orphan layout object should help guide the OP closer to the right solution.

Second, since `aLabel` and `aButton` are not added to the `pane`, they become orphan objects too, and, thus, are invisible when the frame is made visible, as shown in Figure 2.2(a).

[9] `https://forums.oracle.com/forums/thread.jspa?messageID=5698221`



(a) Widgets Invisible (b) Widgets Visible

Figure 2.2: `JFrame` in Listing 2.2 before and after the fix for orphan widgets.

Listing 2.3: Patch for Lisitng 2.2.

```
1 // Updated the coordinates for the button
2 layout.addComponent(aLabel, 5, 5);
3 layout.addComponent(aButton, 80, 5);
4 // Constraints for content pane with respect to aButton
5 layout.putConstraint(EAST, pane, 5, EAST, aButton);
6 layout.putConstraint(SOUTH, pane, 5, SOUTH, aButton);
```

Related, our critic also reports a problem of *Container and Layout Content Mismatch* for `pane`. This is because when the pane is made visible, it contains no child widget but its layout contains both `aLabel` and `aButton`. These two objects can be added by calling `pane.add(aLabel); pane.add(aButton);` before line 22. As shown in Figure 2.2(b), the two widgets then become visible.

Third, our critic points out that the code fails to specify the constraints for the right (`EAST`) and bottom (`SOUTH`) edges of the content pane. As a result, the layout manager cannot correctly compute the size for the content pane. This problem is masked by the call to `setSize()` at line 21, but can be revealed by calling `frame.pack()`, which forces the layout manager to compute its size; as shown in Figure 2.3(a), the widgets are clipped. The patch shown in Listing 2.3 can be applied to replace lines 19-20 in Listing 2.2, resulting in the view shown in Figure 2.3(b).

(a) Before (b) After

Figure 2.3: Listing 2.2 before and after the patch for missing constraints.

Listing 2.4: Case 2 (Parent Switching Positioning and Sizing).

```
1 JFrame frame = new JFrame();
2 frame.getContentPane().setLayout(null); ...
3 jPanel1 = new javax.swing.JPanel();
4 jPanel1.setBounds(700, 50, 270, 400);
5 jPanel1.setLayout(null)
6 frame.getContentPane().add(jPanel1); ...
7 frame.getContentPane().add(jLabel4);
8 jPanel1.add(jLabel4);
9 jLabel4.setBounds(700, 70, 180, 14); ...
10 frame.pack()
11 frame.setVisible(true);
```

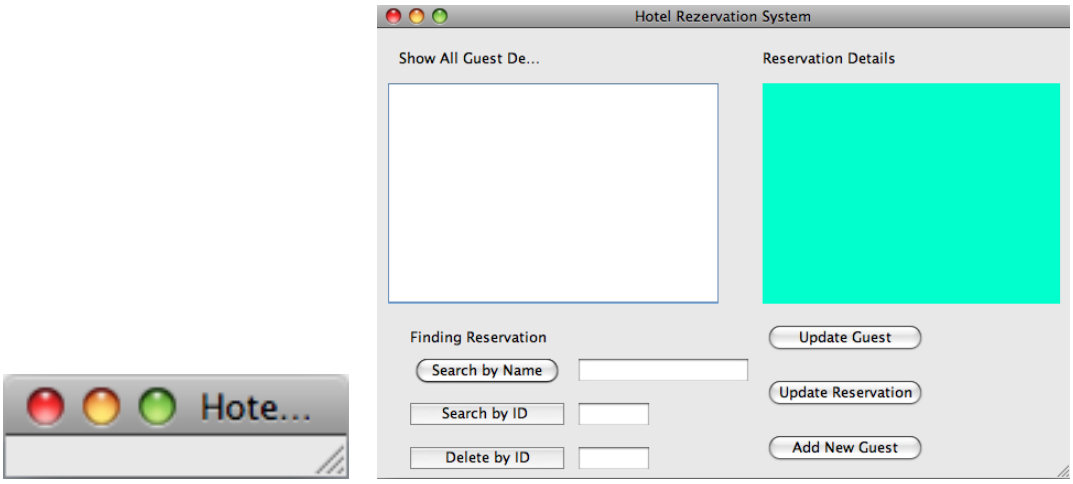
2.3.3 Case 2 (Parent Switching, Positioning and Sizing)

Our critic reveals two problems for Case 2 [10] (Listing 2.4). The first problem is that `jLabel4` is first added to the content pane (line 7) but later to another container `jPanel1` (line 8). As a result, when the GUI tree is made visible, `jLabel4` is visible only under `jPanel1` but not under the content pane. In fact, the OP complained exactly about this. The critic advises the OP to create a new `JLabel`.

The second problem is calling the `pack()` method on a `JFrame` whose layout manager is set to `null`. When the content pane's layout manager is set to `null`, and it does not have a *preferred size* set, the *pack()* method cannot compute the desired size of the window. As a result, the window becomes too small to show its title and content (Figure 2.4(a)). Instead of `pack()`, `setSize()` can be called to explicitly specify a size for the frame (Figure 2.4(b)).

There are cases where a call to `frame.setSize()` and `frame.pack()` appear together, e.g. [11]. These two methods cancel the effect of one another and should not be used together.

[10] `https://forums.oracle.com/forums/thread.jspa?messageID=5714432`
[11] `https://forums.oracle.com/forums/thread.jspa?messageID=5774019`



(a) With `null` layout and `pack()`. (b) Calling `frame.setSize())` to make other GUI widgets visible.

Figure 2.4: The `JFrame` in Case 2 (Listing 2.4).

2.3.4 Case 3 (Dynamic GUIs)

As shown in Listing 2.5 [12], the OP of Case 3 wants to switch widget `c` (line 1) and `lastSelectedLabel` in the container `puzzlePanel`. But the switching is not immediately visible but only after the frame is resized manually.

The process of adding and removing components in the GUI subtree of `puzzlePanel` makes the container invalid and the changes ineffective. Generally, such an issue arises from the dynamic construction of a GUI. The solution for the user is to explicitly tell the Swing framework to redo the layout. It can be done in two ways: by either calling `puzzlePanel.revalidate(); puzzlePanel.repaint();`, or by calling `pack();` on the root widget (`JFrame`) instead of just the `this.invalidate(); this.repaint();` methods in lines 9 and 10. The call to `revalidate()` first invalidates the previously computed size and position of the widgets and recomputes them by performing relayout of the changed container. The `pack()` method recomputes the layout of the whole GUI tree and not just the modified container. The modifications, however, become apparent when the frame resizes because the resizing event forces relayout.

[12] `https://forums.oracle.com/forums/thread.jspa?messageID=5861121`

Listing 2.5: Dynamic GUIs (modified).

```
puzzlePanel.remove(c);
puzzlePanel.remove(lastSelectedLabel);
gbc.gridx=cX;
gbc.gridy=cY;
puzzlePanel.add(lastSelectedLabel,gbc);
gbc.gridx=lX;
gbc.gridy=lY;
puzzlePanel.add(c,gbc);
this.invalidate();
this.repaint();
```

Listing 2.6: A `JWindow` that violates size constraints.

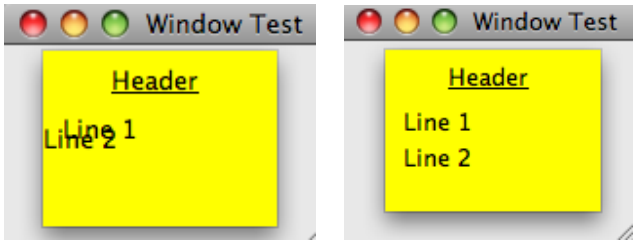
```
private void createAndShowWindow() {
 JWindow win= new JWindow(frame);
 win.setSize(120, 90);
 win.setLocation(90, 50);
 Container cp= win.getContentPane();
 cp.setBackground(Color.YELLOW);
 JLabel lb= new JLabel("<html><u>Header</u></html>");
 lb.setBounds(35,5, 80,20);
 cp.add(lb);
 for (int i=0; i<2; i++) {
  lb= new JLabel("Line "+(i+1));
  lb.setBounds(10,i*20+30, 80,20);
  cp.add(lb);
 }}
 win.setVisible(true);
```

2.3.5 Case 4 (Content Mismatch, Positioning and Sizing)

Our critic reveals two problems from the code in Listing 2.6 [13]. The content pane of the `JWindow` object has a `BorderLayout`. The content pane has all three labels but its `BorderLayout` contains only the last added label (`Line 2`). Hence the first two widgets are positioned using their specified sizes and locations and the last label positioned by the layout manager. As depicted in Figure 2.5(a), the label `Line 2` is positioned at the center location but the other two labels are located at their specified positions.

There can be two solutions to this problem. The first is to use `null` layout by calling

[13] `http://forums.oracle.com/forums/thread.jspa?messageID=9281019`



(a) With layout (b) Without layout

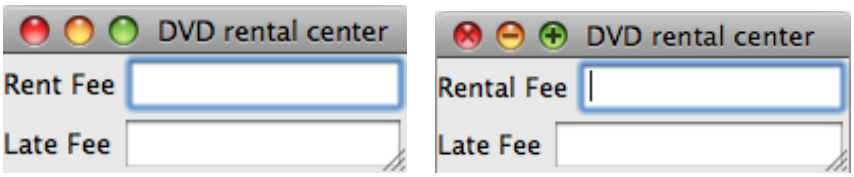
Figure 2.5: The view of `JWindow` before and after the fix.

`cp.setLayout(null)` and completely relying on absolute positioning. Figure 2.5(b) shows the effect of this solution. The second solution, and a better one, is to use only layout managers, without hard-coding sizes and positions.

2.3.6 Case 5 (Table Design, Resizing Conventions)

Listing 2.7: A table implemented using multiple containers (modified).

```
1 public static void main(String[] args) {
2  JFrame frame = new JFrame(“DVD rental center”);
3  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
4  JPanel pane = (JPanel)frame.getContentPane();
5  pane.setLayout(new BoxLayout(pane, BoxLayout.Y_AXIS));
6
7  JLabel rentFee = new JLabel(“Rent Fee ”);
8  JLabel lateFee = new JLabel(“Late Fee ”);
9  JTextField rentFeeField = new JTextField(10);
10 JTextField lateFeeField = new JTextField(10);
11 JPanel p1 = new JPanel(true);
12 JPanel p2 = new JPanel(true);
13 p1.setLayout(new BoxLayout(p1, BoxLayout.LINE_AXIS));
14 p1.add(rentFee);
15 p1.add(rentFeeField);
16 p1.setAlignmentX(JPanel.LEFT_ALIGNMENT);
17 pane.add(p1);
18 p2.setLayout(new BoxLayout(p2, BoxLayout.LINE_AXIS));
19 p2.add(lateFee);
20 p2.add(lateFeeField);
21 p2.setAlignmentX(JPanel.LEFT_ALIGNMENT);
22 pane.add(p2); ...
23 }
```



(a) Labels in the same length. (b) Labels in different lengths.

Figure 2.6: Table design achieved using three `BoxLayouts`.

The code for Case 5 is shown in Listing 2.7 [14]. Although it gives an illusion of a table-like view as shown in Figure 2.6(a), where every widget seems to be positioned properly, this is only coincidental, and our critic gives two criticisms.

First, when the text in `rentFee` is changed from "Rent Fee" to "Rental Fee", as shown in Figure 2.6(b), the two labels become different in length and the table columns are not properly aligned anymore. In general, a table design that spans multiple containers often has alignment issues that are hard to solve. Instead of using multiple containers, the user is advised to use one of `GridLayout`, `GridBagLayout`, or `SpringLayout`.

Second, when the frame is resized, the text fields grow both horizontally and vertically, violating the resizing convention that a text field does not grow vertically. This is because `BoxLayout` is used and the text fields have a large default maximum size that causes them to grow.

2.4 Explanations

Our critic produces an explanation for those API elements that many users commonly find hard to work with, directly in the context where they are used. While explanations are also inherently part of criticisms and recommendations, we have found that explanations can help programmers just by themselves. Programmers often use an API without the full knowledge about its behavior and interaction with other code. They tend to be satisfied as long as the API elements appear to fulfill their needs, ignoring potential side-effects that often become visible later in the development. Explanations are useful in communicating

[14] `https://forums.oracle.com/forums/thread.jspa?messageID=5790663`

Table 2.2: List of helpful explanations discovered in the forum. (D: Direct, I: Indirect, T: Total)

API Explanation Rules	**D / I / T**
Behavior of Null Layout	9 / 3 / 12
Behavior of `GridbagLayout`	5 / 0 / 5
Resizing Behavior of `BorderLayout`	4 / 0 / 4
API Specific Explanations	3 / 0 / 3
Total	**21 / 3 / 24**
Return on Investment	**6**

such non-obvious, subtle behavior of the API. In this way, they facilitate the programmers in reasoning about their code.

In this section, we present some of the useful explanations that we have identified for GUI layout (Table 2.2). Since programmers at different levels of expertise may need explanations with various levels of details, our study also demonstrates how opportunities for explaining API elements can be identified on an as-needed basis, by looking at actual forum discussions.

2.4.1 Behavior of Null Layout

As discussed in Section 2.3.1, when a container has a null layout, it must use *absolute positioning* to position its children. As a result, when the container is resized, its children will not resize automatically. Several OPs used `null` layout but still expected the child widgets to be resized automatically [15]. Our critic explains this implication of null layout to avoid the potential confusion. In addition, since the use of absolute positioning may adversely affect the appearance of the GUI when ported from one platform to another, the user should be advised about this as well.

2.4.2 Centering Behavior of `GridbagLayout`

`GridBagLayout` is a layout manager [16] that positions its child widgets inside a grid. It allows a child to span multiple rows and columns. The visual properties for each child, such

[15] `https://forums.oracle.com/forums/thread.jspa?messageID=5849188`
[16] `http://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html`

as size and growth, are specified as parameters via a `GridbagConstraint` object. Some OPs were confused as to which constraint parameter causes what visual effect [17]. For example, in the absence of at least one non-zero value for `weightx` (`weighty`) for a column (row), that column (row) will not grow with the enclosing container. In the absence of a fill property, a child component at each cell will not grow with the cell, leaving an empty gap when the cell grows bigger. Explaining these can be very helpful for understanding the behavior of the API client code.

2.4.3 Resizing Behavior of `BorderLayout`

A `BorderLayout` positions its children in five pre-defined locations: center, north, south, east, and west. Some novices can be puzzled by its exact behavior. For example, the element in the center takes all of the empty space and ignores the size property explicitly set by the user when resized. As the window becomes smaller, the center widget also grows smaller and eventually gets clipped. Only when there is no more space available for the center widget, do the widgets in the south and then in the north get clipped in the vertical axis and the widgets in the west and then the east get clipped in the horizontal axis. Such explanations can be helpful for the programmer to understand `BorderLayout` properly and also in deciding if the layout is the right choice [18].

2.4.4 API Specific Explanations

Some API elements need to be explained to help programmers understand certain behavior of the API client code. For instance, the call to `setMaximumSize()` on a basic widget such as `JButton` and `JLabel` can cause visual problems on a different platform where the maximum size set may be smaller than the required size [19]. Furthermore, setting the preferred size of one component has the surprising side effect of forcing all components in a `GridLayout` to have the same size [20]. Explaining these API elements can be useful for understanding the

[17] `https://forums.oracle.com/forums/thread.jspa?messageID=5743612`
[18] `https://forums.oracle.com/forums/thread.jspa?messageID=5849282`
[19] `http://forums.oracle.com/forums/thread.jspa?messageID=5698635`
[20] `http://forums.oracle.com/forums/thread.jspa?messageID=5828313`

Table 2.3: Classification of recommendations discovered in the forum.

Description	**Total**
Generic Recommendations	**43**
Syntax-Based Recommendations (Confusing APIs: 5 / Lite Context: 6)	**11**
State-Based Recommendations (Unused Features: 7 / Alternative Design: 3)	**10**
Total	**64**
Return on Investment	**12.8**

API client code.

2.5 Recommendations

What differs an expert from a novice programmer is the amount of information they have about the framework and API. To help novices seek API information, our critic recommends relevant API elements and documentation within the programming context. With enough of the programming context taken into account, the tool can make more precise recommendations on the use of API elements. It can also present competing solutions so that the user can choose the right one based on his or her needs. A key is the capability to infer the programmer's intent from the API client code. However, even with less knowledge of the programming context, our critic should still be able to make generic recommendations, just enough to push novices toward the right direction when they get stuck with their code. In both cases, recommendations bring information where the user needs it the most. Table 2.3 shows the classification of recommendations according to the programming context. We discuss them in order of ease of implementation.

2.5.1 Generic Recommendations

Generic recommendations are applicable even when there is not a proper programming context. Novice programmers often have problems in mapping requirements to the relevant

API elements. For example, consider the following message from the forum [21]:

"Hi, im [i am] quite new to java, and just want to know, how to layout labels and buttons onscreen, [... elided]. If someone could print some sample code with a sample label and layout details that would be a great help. [...]"

Many such problems can be solved by reading documents that are already available online, but which need to be brought to the programmer's attention through recommendation, for instance, the Swing layout tutorial [22]. CriticAL can recommend such documents without any context. We found 43 instances of problems that can be helped by recommending a generic document containing a list of *How-Tos* about the layout API such as "How to use layout managers", "How to achieve absolute positioning", and "How to combine multiple layout managers". Such recommendations should be very useful for novices who do not have much conceptual knowledge of the API [41].

2.5.2 Syntax-Based Recommendation

CriticAL can provide recommendations based on the syntax of the API methods used in the code. We have identified two useful syntax-based recommendations in the forum.

Confusing API Elements

Sometimes users may get confused in choosing the correct API method when the framework contains multiple methods with closely related functionalities. For instance, `JLabel` allows the use of `setAlignmentX()`, `setHorizontalAlignment()`, and `setHorizontalTextPosition()` methods, which are all related to alignment or position, thus confusing the users, e.g. [23] (see Section 1.1.1 for details). CriticAL makes a recommendation of all three methods after detecting the presence of anyone in the group. The recommendation also explains the situation where each method is suitable.

[21] `https://forums.oracle.com/forums/thread.jspa?messageID=5833295`
[22] `http://docs.oracle.com/javase/tutorial/uiswing/layout/`
[23] `http://forums.oracle.com/forums/thread.jspa?messageID=5827139`

Layout Recommendation in Lite Context

It is sometimes possible to infer a user's partial requirements based on the API elements used in his code. Based on such inferences, a potentially applicable layout manager can be recommended. For instance, when a user has multiple components in a container, one of which is a `JTextArea`, we can recommend the `CENTER` location of `BorderLayout` for the text area on the basis that it can grow in both directions, e.g., [24]. Such a recommendation would also list other possibilities such as `BoxLayout` and `GridLayout` in case the user wants the text area to grow proportionally with other widgets, but definitely not `FlowLayout`, which keeps components at their preferred sizes. Furthermore, if the user wants to partition the container disproportionately, then a `GridBagLayout` can be recommended.

2.5.3 State-Based Recommendations

Based on program states, CriticAL can capture the programming context more concretely and provide more precise recommendations.

Recommending Unused Features

CriticAL can be configured to recommend common features used in GUI programming based on the current program state. For instance, by recommending adding a border to the `JPanel` in [25], our CriticAL would help solve the problem of the user who wants to shift all child components in a container to the right by a small distance. Such recommendations help programmers discover API features potentially unknown to them.

Alternative Design

APIs are designed to solve a particular set of problems. It is always beneficial to use API elements in ways that they are intended. The `GridBagConstraint`s used in lines 2-6 and 8 of Listing 2.8 [26] is essentially trying to achieve the behavior of a `FlowLayout` with left

[24] `http://forums.oracle.com/forums/thread.jspa?messageID=9201990`
[25] `http://forums.oracle.com/forums/thread.jspa?messageID=5716439`
[26] `https://forums.oracle.com/forums/thread.jspa?messageID=5729204`

Listing 2.8: Using `GridBagLayout` where `FlowLayout` suited better.

```
1 protected void relayout() { // Complicated way
2  final GridBagConstraints gbc = new GridBagConstraints();
3  gbc.anchor = GridBagConstraints.LINE_START; // Start left
4  gbc.weighty = 0.0; // Grow with a factor of 0
5  gbc.fill = GridBagConstraints.NONE; // Do not fill extra space
6  gbc.insets = new Insets(0, 10, 0, 10); // External padding ...
7  for (int i = 0; i < panels.size();i++) {
8   gbc.gridx = i; // Single line, increase column when adding
9   rootPanel.add(panels.get(i), gbc); // Add element with constraint
10  }...}
11 protected void relayout() { // Simple alternative
12  rootPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
13  for (int i = 0; i < panels.size();i++) {
14   rootPanel.add(panels.get(i)); // No constraint needed
15  } ...}
```

alignment (lines 12-15) for `rootPanel`. CriticAL could be configured to recommend the easier, alternative solution of `FlowLayout`.

2.6 Discussion

In this section, we summarize our observations resulting from this study as answers to our research questions.

As shown in Figure 2.1, 117 of 134 layout-related API discussions can be helped by our critic; there are only 17 truly requirements specific problems, which our critic cannot help since it has no knowledge of the unique user requirements. The data in Figure 2.1 also indicate that all three forms of critiques are needed, and that recommendations and explanations are at least equally important as criticisms for supporting programmers. Overall, this study shows that our critic can be very useful in helping with using APIs.

To be useful, the critic must be equipped with high-quality API usage rules to accurately anticipate a user's goals and address his or her needs. This study supplies the specific API usage rules that can be used in the critic. In addition, at a level higher than the specific rules, we can categorize them into the following common sources:

- Internal consistency rules derived from pre-/post-conditions as well as invariants for

an API;

- Common expectations on program behavior, such as the requirement that a text field and a button normally should not grow vertically;
- Requirements for some common user tasks, such as creating an m x n table, which can be inferred from program states;
- Additional solution procedures, or potentially surprising information, that are commonly used together with solutions already present in the API client code.

Interestingly, we find that the discovered API usage rules are nothing more than the all too familiar preconditions, postconditions, and invariants. Since none of these rules appear to be overly complicated, we conclude that raising programmers' awareness of these rules is the key for improving API usability.

To help measure how often a problem recurs and an API usage rule can be applied, we have calculated a rate of Return On Investment (ROI) for each of the three kinds of critiques in Tables 2.1, 2.2, and 2.3. When resources are limited, the ROIs can be used to prioritize the list of API usage rules to determine which subset should be implemented first.

2.7 Threats to Validity

The results reported in this chapter are based on classifying the layout-related discussions in the Swing Forum. Since the layout API has a strong flavor of a tree data structure, other APIs may exhibit different proportions for the three kinds of critiques. Nevertheless, we anticipate that our research method can be applied to study other frameworks and APIs.

In this study, we identified API usage rules and categorized the discussion threads according to whether they can be helped by these rules or not. These are done solely based on our own interpretation of the forum discussions. Since we cannot directly interview the original poster, we have to assume certain facts about the code and sometimes, an OP's intention. Hence, there is a danger that we may have misinterpreted the situation. We may not have addressed all of the concerns of the original poster in our derived rule set. However, both of these concerns have been mitigated by the good amount of efforts that

we have put into this study and by using another rater (Dr. Hou) throughout the process of coding and classification. Furthermore, our past experience with the Swing framework and its forums [34] can also help.

When calculating ROIs for API rules, we have merged a few rules to simplify the presentation. This is because these rules are logically related. Nevertheless, the presented ROI gives some insights as to the effectiveness and the applicability of the API rules. It should also be noted that the 150 discussions studied are only a very small subset of all the Swing Forum discussions. There are good reasons to believe that there are more cases in the forum that these rules can be applied to.

2.8 Conclusion

To assess to what extent our critic can help solve practical API usage problems and what kinds of API usage rule can be formulated, we manually analyzed 150 discussion threads from the Java Swing forum, from which we created three sets of API usage rules related to the layout logic. We categorized the discussion threads according to how they can be helped by the critic into criticisms, explanations, and recommendations. We illustrate these API usage rules with concrete examples. We find that all three kinds of critiques are useful and justified, API problems of the same nature appear repeatedly in the forum, and API problems of the same nature can be addressed by implementing a new API usage rule. We describe the nature of the API usage rules and how they can be checked. We also discuss the kind of code behavior inference that is needed in order to make the critic smarter and more powerful and the tradeoffs.

Chapter 3

The Design of CriticAL

In this chapter, we will discuss the detailed design of the CriticAL framework. We will formalize the notion of a symbolic object and define operations on it. The CriticAL framework is an interpreter that works on top of the Jimple intermediate representation [61] by manipulating the state of the symbolic objects. We will specify the semantics of execution of each statement using a semi-formal notation. Note that this notation only serves as a means for rigorously communicating our concepts behind the framework.

The CriticAL framework has been designed as an Eclipse plugin that can be further extended by developing new plugins within the Eclipse platform. Figure 3.1 depicts the major components of CriticAL and its plugin architecture. On the bottom layer is the Eclipse platform that provides input (Java project) and output (error markers) services to CriticAL. CriticAL relies on its core plugin for supporting the symbolic execution of a client's code. The core plugin needs the SOOT [61] framework to retrieve control flow graphs of methods under analyses. These graphs are constructed on top of the Jimple intermediate representation provided by SOOT. CriticAL also uses the XStream library for XML serialization and de-serialization of Java objects used storing and retrieving the settings of CriticAL. To support an API, CriticAL must be extended through an extension plugin in which the API objects are modeled symbolically with user-defined abstractions. As a first prototype, we implemented the support for the Java Swing API focusing on

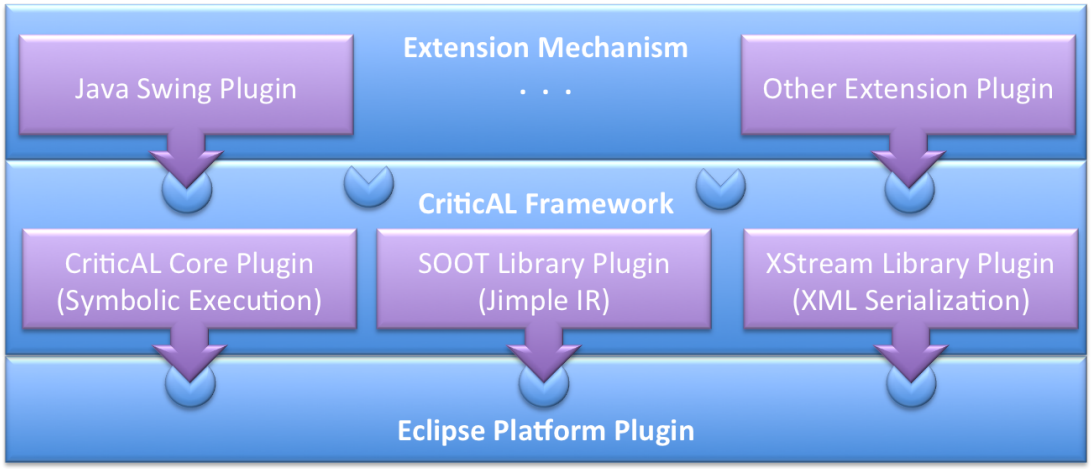


Figure 3.1: The plugin architecture of CriticAL and its major components.

the issues related to layout and GUI composition. CriticAL can support more than one extensions.

3.1 Architectural Overview of the Core

Figure 3.2 depicts the overall architecture for our critic system. The technical foundation for our system is symbolic execution [40], 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 subclasses of these classes.

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 inter-procedural 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 for checking satisfiability and pruning infeasible paths [1].

[1]Note that the implementation of the constraint solving module is a future work. In current implemen-

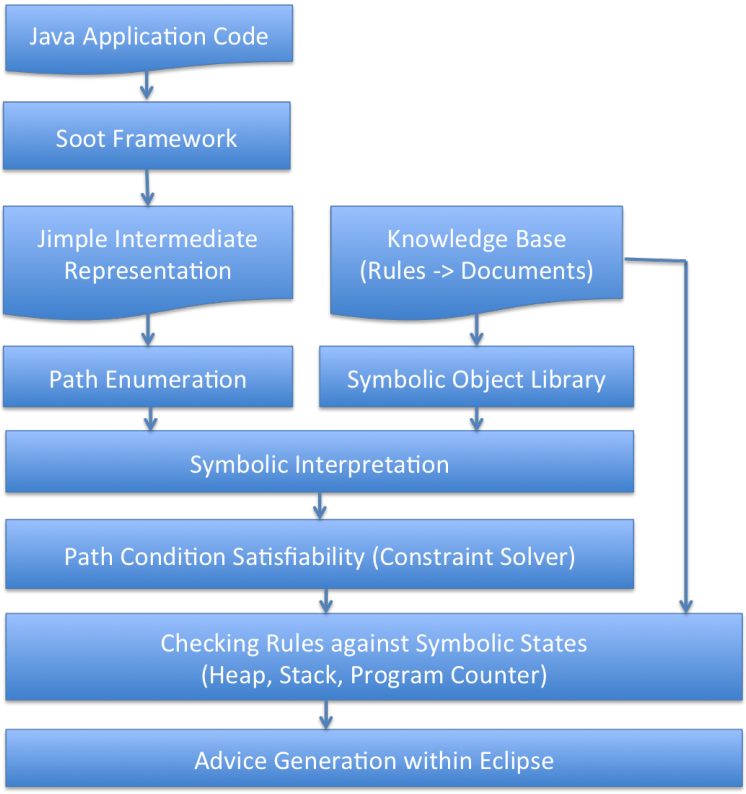


Figure 3.2: Architecture of our critic system.

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. The points of interest can be configured as per the need of analysis.

tation, we conservatively assume that a predicate may be evaluated to true as well as false and generate two paths as a result of executing a conditional statement.

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 effects of the GUI events on the program state [2]. This technique has also been used by others in the generation of GUI test-cases [24, 45].

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 3.2. 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.

3.2 Modeling Symbolic Objects

Let S, Φ, and C be the universe of symbolic objects, Jimple expressions (`Value`), and concrete Java objects, respectively; T be the set of Java types; M be a set of maps whose domains and ranges are symbolic objects ($\{\mu : \{k \mapsto v : k \in S \wedge v \in S\}\}$); and $\Psi : \Phi \rightarrow S$ be the symbolic execution environment (we will provide a more detailed definition of execution environment in Section 3.3.2).

Definition 1 (Symbolic Objects)**.** *A symbolic object $s \in S$ is represented by a tuple $\langle \omega, \tau, \phi, \vartheta, \mu \rangle$ where $\omega \in boolean : \{true, false\}$ represents whether the object is non-deterministic, $\tau \in T$ represents the known type for s, $\phi \in \Phi$ represents the Jimple expression that evaluated s, $\vartheta \in C$ represents the known concrete value of s after evaluation of ϕ (used for primitive types such as int and float), and $\mu \in M$ represents a property-value mapping, which stores properties of the symbolic object (e.g. a symbolic member field mapped to a symbolic value).*

The first three elements in Definition 1 provide meta-information about a symbolic

[2] Note that in the current implementation, we do not take account of the effect of one GUI event on another.

object and the last two represent the value or the state of the symbolic object. Note that ω is *true* when the object is not created within the symbolic execution environment but received from the external environment such as user inputs or parameters of an entry method whose starting value cannot be determined statically. For these cases, only the type information of the symbolic object is assumed from Java's type system but the value is kept open conservatively and represented as a path condition in predicates that involve these values.

3.2.1 Modeling Non-Primitive Types

Let us illustrate the structure of a non-primitive symbolic object through an example. Consider an instruction that creates a `JFrame`: `[JFrame frame = new JFrame("Test");]` with the title `"Test"`. The symbolic execution environment would look like the following:
$\Psi : \{frame \mapsto \langle false, JFrame, NewExpr, \emptyset, \{title \mapsto Test\}\rangle, \ldots\}$,
where `NewExpr` is the Jimple representation for the `new` expression in Java. Note that both the `title` field and the `Test` value are symbolic string objects. (See Section 3.2.3 for details on representing a contiguous block of memory such as a `String`.) Also note that the concrete value of the symbolic `frame` object is null ($\emptyset$) because `JFrame` is a non-primitive type.

3.2.2 Modeling Primitive Types

Let us consider another example: `[int a = 1; int b = 2; int c = a+b;]`. In this case, the symbolic execution environment will look like the following:
$\Psi : \{a \mapsto \langle false, int, IntConstant, 1, \emptyset\rangle, b \mapsto \langle false, int, IntConstant, 2, \emptyset\rangle,$
$c \mapsto \langle false, int, AddExpr, 3, \emptyset\rangle\}$.
Note that `IntConstant` and `AddExpr` (`a+b`) are Jimple representations for an integer constant and an add expression, respectively.

3.2.3 A Completeness Argument

Having defined symbolic object, we need to ensure that such a representation can indeed model the state of all possible Java objects. We will present Theorem 1 to address this concern.

Theorem 1 (Completeness Theorem)**.** *Definition 1 can model the state of all Java objects.*

Proof of Theorem 1. An object in Java can be represented using either contiguous blocks of memory or non-contiguous blocks. To prove the theorem, we will show that Definition 1 can model both cases without any loss of information.

Case 1 - Contiguous Block: In Java, contiguous blocks represent either primitive types such as char, int, and float or an array. We have already shown how primitive values are modeled without a loss of information in Section 3.2.2. An array can be modeled using indices as keys and array elements as values in the property-value map of a symbolic object. For instance, consider the following Java statement `[int[] array = new int[2]{10,11};]`. The `array` object can be modeled as: $\langle$`false, int[], NewArrayExpr,` $\emptyset$`, {dimension` $\mapsto$ `1, length`$_0$ $\mapsto$ `2, 0` $\mapsto$ `10, 1` $\mapsto$ `11}`$\rangle$. Note that even though concrete values are used here, they are, in fact, primitive symbolic objects representing the concrete values. Similarly, multi-dimensional arrays can be represented as a deflated single dimensional array. For such a representation, accessing a location, say $a[row][col]$, can be simply achieved by $a[col + row * length]$. To generalize, for an array of n dimensions, and length of each dimension represented as $l_0, l_1, \ldots, l_k, \ldots, l_{n-1}$, accessing a location $a[i_0] \ldots [i_k] \ldots [i_{n-1}]$ can be achieved by $a[i_{n-1} + i_{n-2} * l_{n-2} + \ldots + i_k * l_{n-k} + \ldots + i_0 * l_0]$.

Case 2 - Non-contiguous Block: Non-contiguous blocks in Java represent an object heap. Every object in the heap will have its own block and the container can access its children through the associated member fields. Clearly, the property-value map of our symbolic object models the object heap without any loss of information. We saw an example in Section 3.2.1 where the `JFrame` had a field called `title` associated to a symbolic string object `Test`. □

3.3 Interpreter

CriticAL is essentially an interpreter that executes Jimple instructions of SOOT. When a project is analyzed, CriticAL first loads all of the classes in the project in SOOT. SOOT then builds a stackless, three-address representation of the Java byte code, which is also known as the Jimple Intermediate Representation (IR). For each non-API method executed by CriticAL, it retrieves the corresponding control-flow graphs and constructs execution paths. The Jimple instructions in each path are then symbolically executed. Hence, for the interpreter to work, we need to support all of the Jimple instructions.

Jimple IR makes static analysis easy by reducing 256 Java Virtual Machine's (JVM) byte code instructions (51 reserved for future use) to 15 Jimple instructions (e.g. if statement, assignment statement, and so on) and 36 operations (add, subtract, and so on). Each Jimple operation has a symbolic counterpart in CriticAL with identical semantics. Figure 3.3 shows the mapping between Jimple IR and CriticAL's counterpart. Note that in the Jimple representation, the `true` boolean value is represented by the integer `1` and `false` by `0`. We will now present the translation and execution semantics.

3.3.1 Translation of Expressions

Translation of Jimple expressions to CriticAL expressions is straightforward since every Jimple expression is mapped to CriticAL's expressions. For instance, an add operation between two integer constants in Jimple IR is interpreted as an addition of two symbolic integer constants in CriticAL. Figure 3.4 presents the semantics of translation. Note that the translation allows us to model the expression trees in object-oriented design. Otherwise, we have to write a long procedural code that switches between different expression types.

3.3.2 Symbolic Execution Environment

So far, we have treated the execution environment (Ψ) as a function whose domain is a set of Jimple values and range is a set of symbolic objects. The execution environment, in fact, represents the execution stack and is modeled as a sequence of stack frames in the

Jimple Exprs		CriticAL's Counterpart
Constants		
IntegerConstant	→	ConstInteger
LongConstant	→	ConstLong
ClassConstant	→	ConstClass
NullConstant	→	ConstNull
DoubleConstant	→	ConstDouble
FloatConstant	→	ConstFloat
Binary Expressions		
EqExpr (==)	→	ExprEq
GeExpr (>=)	→	ExprGe
GtExpr (>)	→	ExprGt
LeExpr (<=)	→	ExprLe
LtExpr (<)	→	ExprLt
NeExpr (!=)	→	ExprNe
AddExpr (+)	→	ExprAdd
AndExpr (&)	→	ExprAnd
CmpExpr (cmp)	→	ExprCmp
CmpgExpr (cmpg)	→	ExprCmpg
CmplExpr (cmpl)	→	ExprCmpl
DivExpr (/)	→	ExprDiv
MulExpr (*)	→	ExprMul
OrExpr (|)	→	ExprOr
RemExpr (%)	→	ExprRem
ShlExpr (<<)	→	ExprShl
ShrExpr (>>)	→	ExprShr
SubExpr (-)	→	ExprSub
UshrExpr (>>>)	→	ExprUshr
XorExpr (~)	→	ExprXor
Unary Expressions		
LengthEpr (lengthof array)	→	ExprLength
NegExpr (-var)	→	ExprNeg
Method Call Expressions		
DynamicInvoke	→	InvokeDynamic
InstanceInvoke	→	InvokeInstance
StaticInvoke	→	InvokeStatic

Jimple Exprs		CriticAL's Counterpart
Object Creation Expressions		
NewExpr (new T())	→	ExprNew
NewArrayExpr (new T[])	→	ExprNewArray
NewMultiArrayExpr (new T[][])	→	ExprMultiArray
References		
ArrayRef (array[i])	→	RefArray
CaughtExceptionRef (catch)	→	RefCaughtException
InstanceFieldRef (a.field)	→	RefInstanceField
ParameterRef (param0)	→	RefParameter
StaticFieldRef (T.field)	→	RefStaticField
ThisRef (this)	→	RefThis
Other Expressions		
CastExpr ((T)var)	→	ExprCast
InstanceOfExpr (o instanceof T)	→	ExprInstanceOf

Jimple Statements		CriticAL's Counterpart
AssignStmt	→	StmtAssign
BreakpointStmt	→	StmtBreakpoint
EnterMonitorStmt	→	StmtEnterMonitor
ExitMonitorStmt	→	StmtExitMonitor
GotoStmt	→	StmtGoto
IdentityStmt	→	StmtIdentity
IfStmt	→	StmtIf
InvokeStmt	→	StmtInvoke
LookupSwitchStmt	→	StmtLookupSwitch
NopStmt	→	StmtNop
RetStmt	→	StmtRet
ReturnStmt	→	StmtReturn
ReturnVoidStmt	→	StmtReturnVoid
TableSwitchStmt	→	StmtTableSwitch
ThrowStmt	→	StmtThrow

Figure 3.3: Jimple IR and CriticAL's counterparts.

implementation. Let us build on the previous definition of execution environment to include this behavior.

Definition 2 (Stack Frame)**.** *A stack frame F is represented by a tuple $\langle C, A_\mu, M_\mu \rangle$ where $C : \langle s, c, m \rangle$ represents the context where $s \in InvokeStmt \cup AssignStmt$ (call site), $c \in$ Resolved Classes, and $m \in$ Resolved Methods; $A_\mu : \{Parameter \mapsto Argument\}$, where $Parameter \equiv ParameterRef \cup ThisRef$ and $Argument \subseteq S$; and $M_\mu : \{Local \mapsto S\}$.*

A stack frame models the execution environment of a method. The first element of the tuple represents the context of the method (C). A context is modeled by three elements:

$Vars \equiv$ Jimple Local Variables $\bigcup$ Static Field References
$Exprs \equiv$ Jimple Expressions
$Refs \equiv$ Jimple References
$Invks \equiv$ Jimple Invoke Expressions
$Consts \equiv$ Jimple Constants
$Value \equiv Vars \bigcup Exprs \bigcup Refs \bigcup Invks$

$CExprs \equiv$ CriticAL Expressions
$CRefs \equiv$ CriticAL References
$CInvks \equiv$ CriticAL Invoke Expressions
$CConsts \equiv$ CriticAL Constants
$S \equiv CExprs \bigcup CRefs \bigcup CInvks \bigcup CConsts \bigcup$ Other Symbolic Objects
$\Psi : Vars \rightarrow S \equiv$ Symbolic execution environment

$T_s : Value \times \Psi \rightarrow S$, for converting any Jimple expression to symbolic object
$T_f :$ Jimple Field $\times \Psi \rightarrow S$, for any non-expression to symbolic object, e.g., fields
$T_t :$ Java Type $\rightarrow$ CriticAL Type, for Java's type system to CriticAL's type system

$T_s[\![Consts, \Psi]\!] = CConsts,$ for Jimple constants
$T_s[\![Vars, \Psi]\!] = \Psi(Vars),$ for Jimple variables and static field references
$T_s[\![\star\phi, \Psi]\!] = \star T_s[\![\phi, \Psi]\!],$ for $\star \in$ unary expressions
$T_s[\![\phi_1 \star \phi_2, \Psi]\!] = T_s[\![\phi_1, \Psi]\!] \star T_s[\![\phi_2, \Psi]\!],$ for $\star \in$ Binary expressions

$T_s[\![T.m(\langle p_1, p_2, \ldots, p_n\rangle), \Psi]\!] = T_t[\![T]\!].m(\langle T_s[\![p_1, \Psi]\!], T_s[\![p_2, \Psi]\!], \ldots, T_s[\![p_n, \Psi]\!]\rangle)$, for static invoke expressions
$T_s[\![o.m(\langle p_1, p_2, \ldots, p_n\rangle), \Psi]\!] = T_s[\![o, \Psi]\!].m(\langle T_s[\![p_1, \Psi]\!], T_s[\![p_2, \Psi]\!], \ldots, T_s[\![p_n, \Psi]\!]\rangle)$, for instance invoke expressions

$T_s[\![\text{new } T, \Psi]\!] = \text{new } T_t[\![T]\!]$, for new expressions
$T_s[\![\text{new } T[length], \Psi]\!] = \text{new } T_t[\![T]\!][T_s[\![length, \Psi]\!]]$, for new array expressions
$T_s[\![\text{new } T[l_0]\ldots[l_n], \Psi]\!] = \text{new } T_t[\![T]\!][T_s[\![l_0, \Psi]\!]]\ldots[T_s[\![l_n, \Psi]\!]]$, for new multi-dimensional array expressions

$T_s[\![o \text{ instanceof } T, \Psi]\!] = T_s[\![o, \Psi]\!]\ instanceof\ T$, for instanceof expression

$T_s[\![a[i_0]\ldots[i_{n-1}], \Psi]\!] = T_s[\![a, \Psi]\!][T_s[\![i_0, \Psi]\!]]\ldots[T_s[\![i_{n-1}, \Psi]\!]]$, for array references
$T_s[\![o.f, \Psi]\!] = T_s[\![o, \Psi]\!].T_f[\![f, \Psi]\!]$, for field references
$T_s[\![Refs, \Psi]\!] = CRefs$, for any other references

Figure 3.4: Translation semantics of Jimple expressions to CriticAL expressions.

the statement that called the method associated with the frame, the resolved class for the method, and the resolved method itself. We use Class Hierarchy Analysis (CHA) [9] to resolve dynamic dispatch for the receiver type of a method. Given a method, and a target class, the CHA algorithm conservatively determines a set of possible target methods that could be executed for the call. Since we perform symbolic execution where an object instantiated during the execution will have a known concrete type, the CHA algorithm returns only one target method for such a case. For an object that is not initialized within the execution environment (e.g. user inputs), CHA may resolve to more than one target method. In such a case, each method is expanded and executed conservatively in a separate execution environment resulting in more than one execution path. The whole execution environment and associated symbolic objects are efficiently cloned for each execution path (see Section 3.7 for discussion on a lazy cloning strategy for efficiency).

The second element of the tuple represents a map for binding parameters to arguments (A_μ). The receiver object of the method is also considered as one of the arguments of the method in our definition.

The third element of the tuple maps the Jimple local variables used in the method to the symbolic objects (M_μ).

Definition 3 (Execution Environment)**.** *The symbolic execution environment* $\Psi : \Phi \rightarrow S$ *is represented internally as a tuple containing a binding between static fields and symbolic objects and a sequence of stack frames,* $\langle G_\mu, F_0, \ldots, F_k, \ldots, F_{n-1} \rangle$*, where* $G_\mu : \{$*Static Field Reference* $\mapsto S\}$*,* F_{n-1} *represents the stack top for a call depth of* n *methods, such that* $\Psi(\phi) = G_\mu[\phi]$ *for all* $\phi \in$ *Static Field Refs and* $\Psi(\phi) = F_{n-1}.M_\mu[\phi]$ *for all* $\phi \in Value -$ *Static Field Refs.*

The lookup for static field references is delegated to G_μ and for other Jimple values such as local variables, delegated to the top stack frame of the execution environment, i.e. $F_{n-1}.M_\mu$. Whenever a new method is executed, a new stack frame is created in the execution environment and the mappings between parameters and arguments are established. As an execution semantics of a return statement of the recently executed method, the top stack

frame is popped from the environment. If the method is returning a value, then the return value (symbolic object) is mapped to the expecting Jimple variable in the next stack frame.

3.4 Execution Semantics

In this section, we will discuss the formal execution semantics of CriticAL for all of the Jimple statments. Jimple statements are first translated to CriticAL's counterparts before execution. We will exclude the discussion about translation semantics for brevity as they are straightforward (see Figure 3.3) and concentrate on their execution semantics in this section. Before delving into the details of execution semantics, let us define a program state.

Definition 4 (Program State)**.** *The state of a program during a symbolic execution is represented by the pair* $\langle \Psi, P \rangle$*, where* Ψ *is the execution environment and* P *is the path condition represented as a set of predicates that hold in the path.*

Note that the arguments to a predicate represent free variables or a concrete value (further discussed in Section 3.5). The execution of a statement may modify Ψ and/or P.

3.4.1 Identity Statement

In Jimple IR, the identity statement assigns `ThisRef` and `ParameterRef` to Jimple local variables [3]. For a non-static method `m(int a, int b)`, the Jimple translation (close but not exact) looks like the following:

```
this := @ThisRef;
a := @ParameterRef0;
b := @ParameterRef1;
...
```

Figure 3.5 in Section 3.4.5 shows the translation of a static method in Java to Jimple IR that contains examples of identity statements. Note that Jimple IR is a stack-less representation of Java byte code, which necessitate the use of the identity statement for representing the

[3]Note that we ignore `CaughtExceptionRef` on the right hand side of the identity statement in the current implementation.

receiver object (`this`) and the parameters (`a`, `b`). For a static method, there will be no `this := @ThisRef;` in the translation. Now consider an identity statement `v := Ref`, where `Ref` ≡ `ThisRef` ∪ `ParameterRef`. There are two cases to handle:

Case 1: Entry Methods

An entry method is the method from which a symbolic execution starts. For an entry method ($n = 1$), there is no binding between parameters and arguments in the top stack frame of the execution environment ($n = 1 \wedge \Psi.F_{n-1}.A_\mu = \emptyset$). The execution semantics for such an identity statement is as follows:

$$\langle \Psi, P \rangle, [v := Ref] \Rightarrow_E \langle \Psi[F_{n-1}.M_\mu[v \mapsto \langle true, \tau, Ref, \emptyset, \mu \rangle]], P \rangle \tag{3.1}$$

where $\Rightarrow_E$ represents execution of the identity statement, the `true` value in the binding represents an open symbolic object (non-deterministic), τ is the known type derived from the `Ref` expression, $\emptyset$ represents unknown concrete value, and μ represents the property-value map, which is empty. Note that the binding between the Jimple variable and the newly created symbolic object happens in the top stack frame F_{n-1}. Also note that the corresponding type for the symbolic object is first looked up in CriticAL's extension plugins and then initialized. If none of the extension plugins support the required type, then CriticAL creates a default symbolic object with the given type information.

Case 2: Non-entry Methods

For a non-entry method ($n > 1$), there are bindings between parameters and arguments in the top stack frame. The execution semantics for such an identity statement is as follows:

$$\langle \Psi, P \rangle, [v := Ref] \Rightarrow_E \langle \Psi[F_{n-1}.M_\mu[v \mapsto F_{n-1}.A_\mu[Ref]]], P \rangle \tag{3.2}$$

3.4.2 Assignment Statement

The left hand side of an assignment statement can be one of the four kind of expressions: local variables, static field references, instance field reference, and array references. The right hand side can be any Jimple expressions. We will now present the execution semantics for all four expressions on the left hand side of the assignment statement.

Semantics for Left Hand Side Expressions

Case 1: Assignment to a Local Variable

$$\langle \Psi, P \rangle, [l = \phi] \Rightarrow_E \langle \Psi[F_{n-1}.M_\mu[l \mapsto T_s[\![\phi, \Psi]\!]], P \rangle \tag{3.3}$$

Case 2: Assignment to a Static Field Reference

$$\langle \Psi, P \rangle, [T.f = \phi] \Rightarrow_E \langle \Psi[G_\mu[T.f \mapsto T_s[\![\phi, \Psi]\!]], P \rangle \tag{3.4}$$

Case 3: Assignment to an Instance Field Reference

$$\frac{T_s[\![o, \Psi]\!] \Rightarrow_E s}{\langle \Psi, P \rangle, [o.f = \phi] \Rightarrow_E \langle \Psi[s.\mu[T_f[\![f, \Psi]\!] \mapsto T_s[\![\phi, \Psi]\!]]], P \rangle} \tag{3.5}$$

Case 4: Assignment to an Array Reference

$$\frac{T_s[\![a, \Psi]\!] \Rightarrow_E s}{\langle \Psi, P \rangle, [a[i] = \phi] \Rightarrow_E \langle \Psi[s.\mu[T_s[\![i, \Psi]\!] \mapsto T_s[\![\phi, \Psi]\!]]], P \rangle} \tag{3.6}$$

Semantics for Right Hand Side Expressions

The translation semantics in Figure 3.4 recursively translates expression trees to their leaf elements, however, not all of the leaf-elements may be evaluated as constants. Hence, for non-constant leaf-elements in an expression tree, we need to specify their execution semantics. We will now present case-by-case recursive rules for the evaluation of $T_s[\![\phi, \Psi]\!]$ for such cases.

Case 1: $\phi \in$ **Local Variables**

$$T_s[\![\phi, \Psi]\!] \Rightarrow_E \Psi.F_{n-1}.M_\mu[\phi] \tag{3.7}$$

Case 2: $\phi \in$ **Static Field References**

$$T_s[\![\phi, \Psi]\!] \Rightarrow_E \Psi.G_\mu[\phi] \tag{3.8}$$

Case 3: $\phi \in$ **Unary Expressions**

There are two kinds of unary expressions:

1) Negative Expression:

$$\frac{T_s[\![\phi, \Psi]\!] \Rightarrow_E s}{-T_s[\![\phi, \Psi]\!] \Rightarrow_E \begin{cases} -s \mid (-s).\omega = true & \text{if } s.\omega = true \\ \langle s.\omega, s.\tau, s.\phi, -(s.\vartheta), \emptyset \rangle & otherwise \end{cases}} \tag{3.9}$$

The first case in Rule 3.9 represents an open object s which is not created within the symbolic execution environment. Hence, the negation of such an open object is not concretized (or evaluated). In the second case, however, s is known and its concrete value is used to evaluate the negation expression and a replica of the $-s$ object is created to represent the result.

2) Lengthof Expression:

$$\frac{T_s[\![\phi, \Psi]\!] \Rightarrow_E s, \quad l = \langle true, int, lengthof\ \phi, \emptyset, \mu \rangle}{lengthof\ T_s[\![\phi, \Psi]\!] \Rightarrow_E \begin{cases} s.\mu[length_d] & \text{if } s.\mu[length_d] \neq \emptyset \\ l \mid s.\mu[length_d] = l & \text{otherwise} \end{cases}} \tag{3.10}$$

Note that the value of the dimension d of the array can be obtained from the Jimple expression. The first case in Rule 3.10 implies that the length of the d^{th} dimension is known and already bound to a symbolic value (see Section 3.2.3 for the representation of an array).

If that binding does not exist, then the array object must be open (second case), hence, a new symbolic open value is created to represent the unknown length and the new binding for the length is established in s.

Case 4: $\phi \in$ Binary Expressions

There are two cases for binary expressions:

1) Conditional Expressions:

$$\frac{T_s[\![\phi_1, \Psi]\!] \Rightarrow_E s_1, \quad T_s[\![\phi_2, \Psi]\!] \Rightarrow_E s_2}{s_1 \star s_2 \Rightarrow_E \begin{cases} s_1 \star s_2 \mid (s_1 \star s_2).\omega = true & \text{if } s_1.\omega = true \vee s_2.\omega = true \\ \langle false, boolean, (s_1 \star s_2).\phi, s_1.\vartheta \star s_2.\vartheta, \emptyset \rangle & \text{otherwise} \end{cases}} \tag{3.11}$$

2) Numeric Expressions:

$$\frac{T_s[\![\phi_1, \Psi]\!] \Rightarrow_E s_1, \quad T_s[\![\phi_2, \Psi]\!] \Rightarrow_E s_2}{s_1 \star s_2 \Rightarrow_E \begin{cases} s_1 \star s_2 \mid (s_1 \star s_2).\omega = true & \text{if } s_1.\omega = true \vee s_2.\omega = true \\ \langle false, \tau_h, (s_1 \star s_2).\phi, s_1.\vartheta \star s_2.\vartheta, \emptyset \rangle & \text{otherwise, where } \tau_h \equiv \text{type of } s_1.\vartheta \star s_2.\vartheta \end{cases}} \tag{3.12}$$

Note that τ_h in the second case of Rule 3.12 represents the concrete type resulting from the evaluation of the expression.

Case 5: $\phi \in$ New Expressions

There are three kinds of `new` expressions:

1) New Type Expression:

$$new\ T_t[\![T]\!] \Rightarrow_E \langle false, T, new\ T, \emptyset, \mu \rangle \tag{3.13}$$

creates a new symbolic object after looking up in the registered extension plugins for the new type. If the type is not supported, then a default symbolic object is created with the

type information.

2) New Array Expression:

$$\frac{T_s[\![length, \Psi]\!] \Rightarrow_E l, \quad d_1 = \langle false, int, \phi, 1, \emptyset \rangle}{\begin{array}{l} new\ T_t[\![T]\!][T_s[\![length, \Psi]\!]] \Rightarrow_E \\ \quad \langle false, T, new\ T[length], \emptyset, \mu[dimension \mapsto d_1, length_0 \mapsto l] \rangle \end{array}} \tag{3.14}$$

3) New Multi-Array Expression:

$$\frac{T_s[\![l_k, \Psi]\!] \Rightarrow_E l'_k, \quad d_n = \langle false, int, \phi, n, \emptyset \rangle}{\begin{array}{l} new\ T_t[\![T]\!][T_s[\![l_0, \Psi]\!]] \ldots [T_s[\![l_n, \Psi]\!]] \Rightarrow_E \\ \quad \langle false, T, new\ T[l_0] \ldots [l_{n-1}], \emptyset, \mu[dimension \mapsto d_n, length_0 \mapsto l'_0, \ldots, length_{n-1} \mapsto l'_{n-1}] \rangle \end{array}} \tag{3.15}$$

where the dimension n is obtained from the Jimple expression.

Case 6: $\phi \in$ References

The references that may exist on the right hand side of assignment statements are: `StaticFieldRef`, `InstanceFieldRef`, and `ArrayRef`. Other references (`ThisRef`, `ParameterRef`, and `CaughtExceptionRef`) can only exist on the right hand side of identity statements, which we have already dealt with in Section 3.4.1. Furthermore, `StaticFieldRef` is handled by Rule 3.8. Let us now specify the semantics for the remaining two cases:

1) Instance Field References:

$$\frac{T_s[\![o, \Psi]\!] \Rightarrow_E o', \; T_f[\![f, \Psi]\!] \Rightarrow_E f'}{T_s[\![o, \Psi]\!].T_f[\![f, \Psi]\!] \Rightarrow_E o'.\mu[f']} \tag{3.16}$$

where $o'.\mu[f'] = \langle true, T_{field}, o.f, \emptyset, \mu \rangle$ if $o'.\omega = true$ where T_{field} is the type derived from the field reference Jimple expression. Note that o' is a symbolic object and f' represents

the field f of o'.

2) Array References:

$$\frac{T_s[\![a,\Psi]\!] \Rightarrow_E a',\ T_s[\![i_k,\Psi]\!] \Rightarrow_E i'_k,\ i'_{n-1} + i'_{n-2} * l_{n-2} + \ldots + i'_0 * l_0 \Rightarrow_E \gamma}{T_s[\![a,\Psi]\!][T_s[\![i_0,\Psi]\!]] \ldots [T_s[\![i_{n-1},\Psi]\!]] \Rightarrow_E a'.\mu[\gamma]}$$

where $a'.\mu[\gamma] = \langle true, T_\gamma, a[i_0] \ldots [i_{n-1}], \emptyset, \mu \rangle$ if $a'.\omega = true$ where T_γ is the type derived from the Jimple array reference expression. Note that for an array of n dimensions, the length of each dimension is represented as $l_0, \ldots, l_{n-1}$ and are bound to symbolic values in a'. The length expressions are evaluated using Rule 3.10.

Case 7: $\phi \in$ Casting Expression

Let us deal with a case that has a local variable on the left hand side. This case can be easily generalized to references using Rules 3.4-3.6.

$$\frac{T_s[\![\phi,\Psi]\!] \Rightarrow_E s}{\langle \Psi, P \rangle, [l = (T)\phi] \Rightarrow_E \begin{cases} \langle \Psi[F_{n-1}.M_\mu[l \mapsto s]], P\{T \trianglelefteq s.\tau\} \rangle & \text{if } s.\omega = false \\ \langle \Psi[F_{n-1}.M_\mu[l \mapsto s]], P \rangle \mid s.\tau = T & \text{if } s.\omega = true \wedge T \rhd s.\tau \\ \langle \Psi[F_{n-1}.M_\mu[l \mapsto s]], P \rangle & \text{otherwise.} \end{cases}} \quad (3.17)$$

In Rule 3.17,$L \trianglelefteq R$ means that L is a super-type or exact type of R and $L \rhd R$ means that L is a strict sub-type of R. If s is a known object whose type information is concrete, then we use the type hierarchy information provided by Eclipse's JDT (Java Development Tools) [4] to check the path-condition during the execution. If the path-condition is unsatisfiable CriticAL informs the user about `ClassCastException`. If s is open, then we bind s to the new type T if T is a strict subtype of the known type for s. Note that even though this binding accumulates more precise information about the type of s, it may also introduce unsoundness if the casting should indeed fail for s in the actual run of the program.

[4] `http://www.eclipse.org/jdt/`

Case 8: $\phi \in$ **Instanceof Expression**

$$\frac{\begin{array}{c} T_s[\![o, \Psi]\!] \Rightarrow_E o', \quad b = o' \ instanceof \ T, \\ t = \langle false, boolean, \phi_{true}, 1, \emptyset \rangle, \quad f = \langle false, boolean, \phi_{false}, 0, \emptyset \rangle \end{array}}{T_s[\![o, \Psi]\!] \ instanceof \ T \Rightarrow_E \begin{cases} t, & \text{if } o'.\tau \trianglerighteq T \\ f, & \text{if } s.\omega = false \wedge \neg(o'.\tau \trianglerighteq T) \\ b \mid b.\omega = true & \text{otherwise} \end{cases}} \tag{3.18}$$

We we will discuss **Case 9:** $\phi \in$ **Invoke Expression** as a following separate subsection as it is an important part of CriticAL's execution semantics.

3.4.3 Executing Invoke Expressions

There are three kinds of invoke expressions or method calls: `InstanceInvokeExpr`, `StaticInvokeExpr`, and `DynamicInvokeExpr`. Note that `InstanceInvokeExpr` has three more subtypes (`InterfaceInvokeExpr`, `SpecialInvokeExpr`, and `VirtualInvokeExpr`), nevertheless, we handle all three in the `InvokeInstance` expression of CriticAL.

Note that an extension plugin is a set of types that are implemented as Java classes. The purpose of each of them is to abstractly interpret the behavior of a corresponding API type. Given a method call, CriticAL checks if the method is declared in one of the user created application classes in the project being analyzed. CriticAL expands such a method by pushing a new stack frame in the execution environment and performing parameters to arguments binding for the method. If the method is not declared in one of the application classes, then CriticAL looks up in the registered extension plugins for the Java type that declares the method. The execution semantics of a supported API-method is specified within the method of the Java type, which gets executed for the call. For such an API method call, the extension plugin developer may modify the receiver and the parameters of the method (bound to corresponding symbolic arguments) and return a symbolic value as a part of the execution semantics of the API method.

If the registered plugins do not have support for the method, then CriticAL creates a default open symbolic object as a return value to the method call. Such an open object can also be used to model user inputs statically. We will now present semantics for both executing an unexpanded method call as well as an expanded method call.

1) Unexpanded Method Call:

$$\frac{T_s[\![o,\Psi]\!] \Rightarrow_E o' \,(\text{receiver}), \quad T_s[\![p_k,\Psi]\!] \Rightarrow_E p'_k \,(\text{parameters})}{T_s[\![o,\Psi]\!].m(\langle T_s[\![p_1,\Psi]\!], T_s[\![p_2,\Psi]\!], \ldots, T_s[\![p_n,\Psi]\!]\rangle) \Rightarrow_E \begin{cases} \gamma_{user} & \text{if } m \in T_t[\![o'.\tau]\!] \\ \gamma_{default} & \text{otherwise} \end{cases} \tag{3.19}$$

where γ_{user} is the user-defined return value resulting from the direct execution of the corresponding method in the $T_t[\![o'.\tau]\!]$ class of a registered extension plugin. A good analogy of this mechanism is the way in which Java deals with native methods whose code is not a part of the Java Virtual Machine (JVM) but gets executed when the JVM executes the corresponding method call. Similarly, CriticAL also executes the code directly.

If m is not supported in the corresponding CriticAL's extension plugin class ($T_t[\![o'.\tau]\!]$), then a default open symbolic object is returned ($\gamma_{default} = \langle true, (o'.m(p'_1, \ldots, p'_n)).\tau, o.m(p_1, \ldots, p_n), \emptyset, \mu\rangle$). Note that the semantics for handling a static method is almost the same. The only difference is that there will be no receiver object and CriticAL will look for the corresponding static method in the $T_t[\![T]\!]$ class of a registered extension plugin.

2) Expanded Method Call:

For an expanded method call, the return value is not known until a return statement is executed. Hence, the binding between the expecting local variable and the return value is not established. However, a new stack frame F is pushed with the necessary context

information C and bindings between parameters and arguments A_μ.

$$\frac{T_s[\![o, \Psi]\!] \Rightarrow_E o' \text{ (receiver)}, \quad T_s[\![p_k, \Psi]\!] \Rightarrow_E p'_k \text{ (parameters)}}{\langle \Psi_{n=k}, P\rangle, [l = o.m(p_1, \ldots, p_n)] \Rightarrow_E \langle \Psi_{n=k+1}, P\rangle \mid \Psi_{n=k+1}.F_{n-1}.A_\mu[this \mapsto o', p_1 \mapsto p'_1, p_n \mapsto p'_n]}$$

Note that the semantics for handling static method is almost the same here as well. The only difference is that there will be no $this \mapsto o'$ in $\Psi_{n=k+1}.F_{n-1}.A_\mu$.

3.4.4 Return Statements

The expanded method call as seen in Rule 3.20 does not bind the return value to the expecting local variable of the caller's frame. The return statement performs this binding after popping the top stack frame out of the execution environment.

$$\frac{T_s[\![r, \Psi]\!] \Rightarrow_E \gamma \text{ (return value)}, \quad [l = o.m(p_1, \ldots, p_n)] \text{ (call site)}}{\langle \Psi_{n=k}, P\rangle, [return\ r] \Rightarrow_E \langle \Psi_{n=k-1}, P\rangle \mid \Psi_{n=k-1}.F_{n-1}.M_\mu[l \mapsto \gamma]} \tag{3.20}$$

where the call site can be accessed using $\Psi.F_{n-2}.C.s$. Note that for an entry method there is no call site, hence we only pop the top stack frame without performing any binding [5]. The same is true for `ReturnVoidStmt` which does not return a value. Also note that an `InvokeStmt` encapsulates an `InvokeExpr` which does not return a value and is treated in the same fashion. For a value retuning method, Jimple guarantees that the return value is assigned to a local variable through an `AssignStmt`, even if the code at the Java-level does not have such an assignment.

3.4.5 If Statement

All of the Jimple statements that we have handled so far have only one successor statement in the control-flow graph (except return statements that have none). `IfStmt` has two successors: one when the conditional expression (binary expression) in the `IfStmt` is evaluated to true, and the other when the conditional expression is evaluated to false. If the operands

[5] An entry method is the method from which the symbolic execution started.

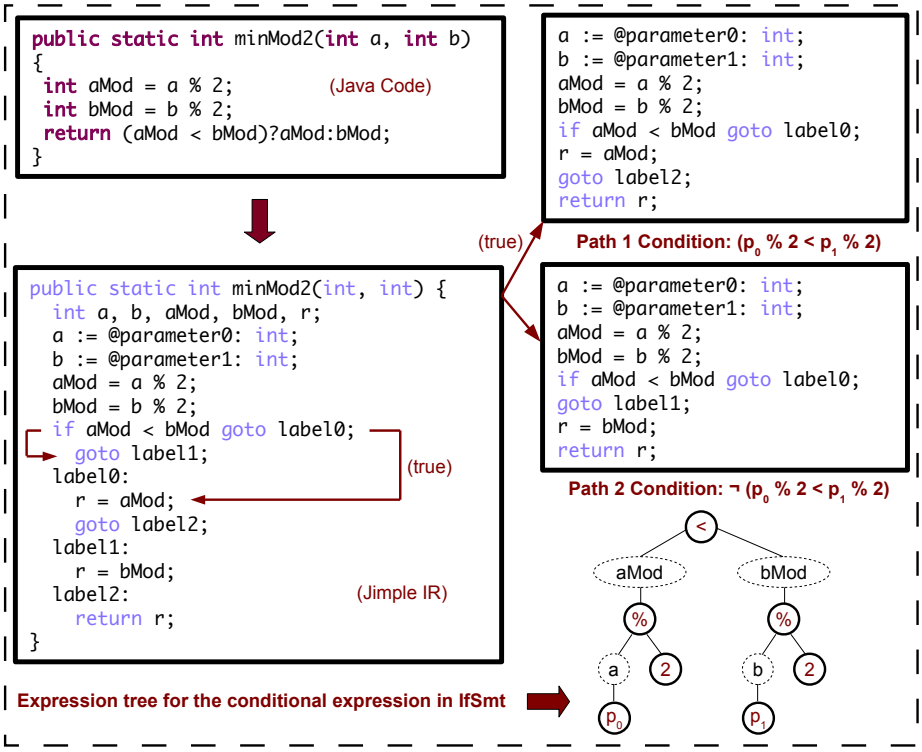


Figure 3.5: Path conditions generated for the `IfStmt` containing open symbolic objects.

of the binary expression are concrete (not open) during symbolic execution, then we will get a concrete result (true or false) easily using the second case of Rule 3.12. However, when one of the operands is open, then the first case of Rule 3.12 applies and we have to rely on a constraint solver to test the satisfiability of the path condition. If the constraint solver says both conditions are satisfiable, then we need to process two paths: the first path will assume the condition is satisfiable and executes the next statement corresponding to the true-branch, the second path will assume the condition is not satisfiable and executes the statement corresponding to the false-branch.

Figure 3.5 illustrates these concepts with the help of a simple method that calculates the minimum mod-2 of the two parameters (`a` and `b`). Path 1 in the figure corresponds to the true-branch and Path 2 corresponds to the false-branch. Note that since the current implementation of CriticAL does not have a constraint solving module, we assume that both paths are feasible. However, in the future implementation, we would repeat this process

and accumulate the path conditions and constraint solve at every open condition in the program. If the constraint solver says that a path condition is unsatisfiable then we assume the path is infeasible and halt the symbolic execution of the corresponding path.

We will now specify the semantics of `IfStmt` formally. Let p_c be the symbolic condition evaluated using Rule 3.12; $if(p_c) \rightarrow_{cf} \{S_T, S_F\}$ represents the control-flow of the if statement to its successor statements corresponding to the true and the false branches, respectively; and $t = \langle false, boolean, \phi_{true}, 1, \emptyset \rangle$ represents the `true` symbolic value. There are two cases:

Case 1: Known Symbolic Condition ($p_c.\omega = false$)

$$\langle \Psi, P \rangle, [if(p_c)], [if(p_c) \rightarrow_{cf} \{S_T, S_F\}] \Rightarrow_E \begin{cases} \langle \Psi, P \rangle, [S_T] & \text{if } p_c = t \\ \langle \Psi, P \rangle, [S_F] & otherwise \end{cases} \tag{3.21}$$

Case 2: Open Symbolic Condition ($p_c.\omega = true$)

$$\langle \Psi, P \rangle, [if(p_c)], [if(p_c) \rightarrow_{cf} \{S_T, S_F\}] \Rightarrow_E \begin{cases} \langle \Psi, P_T \mid P \cup \{p_c\} \rangle, [S_T] & \text{if } sat\ P_T \\ \langle \Psi, P_F \mid P \cup \{\neg p_c\} \rangle, [S_F] & \text{if } sat\ P_F \\ \langle \Psi, P \rangle, [\emptyset] & \text{otherwise (terminate)} \end{cases} \tag{3.22}$$

Note that Ψ in the first two cases of Rule 3.22 is cloned to avoid any side-effects using Algorithm 1 (Section 3.7). Also note that only Ψ is cloned but not the symbolic objects contained in it until a mutate operation is carried out on one of the shared symbolic objects (Algorithm 2, Section 3.7). Section 3.7 presents an efficient cloning strategy to minimize the memory consumption due to cloning and maximize the sharing between clones of an execution environment.

That covers the semantics of all of the important Jimple statements. Note that `LookupSwitchStmt` and `TableSwitchStmt`, variants of Java's *switch-case* statement, are handled using semantics identical to `IfStmt` and are elided in our discussion. Thread synchronization related statements: `EnterMonitorStmt` and `ExitMonitorStmt` correspond to the

Java's synchronized block, which we ignore in the current implementation of CriticAL. Similarly, other remaining statements: `BreakpointStmt`, `GotoStmt`, `NopStmt`, `RetStmt`, and `ThrowStmt` contribute to the control-flow and do not update our execution environment and hence are ignored. Nevertheless, we take into account their effect on the control-flow during execution.

3.5 A Glimpse at the Constraint Solving Module

The main function of the constraint solving module in CriticAL would be to prune infeasible paths. However, with the constraint solving module in place, CriticAL could be used for bounded program verification as well. In this section, we will present a preliminary discussion on the translation process of CiritcAL's symbols to predicate logic (First-Order) that can be constraint solved using a generic constraint solver such as Yices [14] and Choco [6]. We will reuse the example in Figure 3.5 to discuss this process. Here are the rules for translation:

1. An open, leaf symbolic object ($s \mid s.\phi \in \{$`ThisRef`, `ParameterRef`, `StaticFieldRef`, `InvokeExpression`$\}$ will be modeled as a free logic variable. Note that `InvokeExpression` here is an unexpanded method call. Also note that conditional expressions guarantee that either primitive types or references of composite types are being compared. In the case of the `instanceof` expression, containment relations between two types are checked. Assuming Java types can be modeled using sets, there is no fundamental difficulty in representing conditional expression checking subtypes as a predicate checking subset. Similarly, references can be modeled using integers. Most of the constraint solvers support operations on these primitive types and hence can be modeled.

2. Given a conditional expression $\phi_1 \star \phi_2$, we will recursively look up bindings for ϕ_1 and ϕ_2 in Ψ until we reach leaf-elements, thus, constructing an expression tree. For instance, in Figure 3.5, the condition `aMod < bMod` was expanded to the expression

[6] `http://sourceforge.net/projects/choco`

tree by recursively looking up bindings for local variables until leaf elements are encountered. Note that every symbolic object has the corresponding Jimple expression that evaluated it ($s.\phi$) to make this process possible.

3. All of the open-symbolic objects are declared as a free variable, concrete values are used as is, and operators are used as is to construct predicates. In the case of Figure 3.5, the logic model for checking satisfiabitlity at the `IfStmt` for the true-branch (Path 1) would look like the following:

   ```
   int p0, p1            // Free variable declarations
   p0 % 2 < p1 % 2       // Path Condition
   sat                   // Checking satisfiability
   ```

   The constraint solver would answer satisfiable as it can find a satisfying assignment for p_0 and p_1. The solver would answer satisfiable for the model of Path 2 as well. Hence, the constraint solving approach does not yield much over the existing conservative approach of CriticAL for this example because both of them do not prune any path. While this example does not show any advantage of constraint solving, however, constraint solving may become more useful when the same variables are constrained with multiple path conditions.

Note that this is just a glimpse at the future work of the constraint solving module and we do not claim this module as a contribution of this thesis. There may be some details that need to be researched further. We did not invest our effort in the constraint solving module because we have not yet found a convincing example from the Swing forum that would benefit from a constraint solver. The reuse nature of API-client code involve multiple calls to API methods rather than code for implementing low-level algorithms that involves multiple condition checking on the same API object.

3.6 Unrolling Loops

Loops need to be handled specially during symbolic execution. It may be possible to completely execute a loop with the actual number of iterations based on its loop condition if the condition does not involve open symbolic objects. However, most of the critiques could be produced in a small number of iterations. Hence, it may be desirable to unroll a loop up to a small bound to curb path explosion. Several symbolic execution techniques [11, 12, 38] as well as bounded program verification techniques [62] employ this strategy. CriticAL also uses this strategy and unrolls a loop twice in a path, i.e., allows three repetitions of a loop condition in a path (can be configured by a user). Similarly, to tackle against recursive methods, CriticAL allows limiting the depth of the method call. Note that unrolling a loop n times may introduce unsoundness especially when a client code would produce critiques on the $(n+1)^{st}$ iteration, however, the same is true with any symbolic execution techniques that employ similar approach.

Let us illustrate the loop unrolling behavior with an example as shown below:

```
S1; while(C) S2; S3;
```

CriticAL will execute the following three paths for this code assuming that `C` is open:

1. `S1; ¬C; S3;`
2. `S1; C; S2; ¬C; S3;`
3. `S1; C; S2; C; S2; ¬C; S3;`

3.7 Maximal Sharing Strategy

We have previously discussed in Section 3.4.5 that the execution environment (or Stack) needs to be cloned after encountering an open condition in an `IfStmt`. We assumed that all of the stack frames and corresponding symbolic objects are cloned to avoid any side-effects from sharing them in the multiple stacks. However, in the actual implementation, even though the stack and stack frames are cloned, symbolic objects are shared by multiple

stacks until one of the execution stacks updates the objects. Just before the update, the corresponding symbolic objects are cloned for the executing stack. We use proxy patterns with lazy initialization to achieve this maximal sharing behavior.

The design of this strategy requires maintaining two kinds of reverse pointers in symbolic objects:

Reverse Stack Pointer Every symbolic object maintains a set of reverse stack pointers to the stack frames that bind it to Jimple local variables and is represented by the following tuple: $\langle F_*, Value \rangle$, where $F_* \in \Psi$ and $Value \equiv Jimple\ Value$. Using reverse stack pointers a symbolic object can find out all of the stack frames that share it.

Reverse Heap Pointer Every symbolic object maintains a set of reverse heap pointers to the symbolic objects that contain it and is represented by the following tuple: $\langle P_t, S \rangle$, where $P_t \in \{\text{“}Key\text{”}, \text{“}Value\text{”}\}$ and S is the container object which contains the symbolic object as a key or value in its μ or as an element of its custom data structure. Using reverse heap pointers a symbolic object can find all of its containers in the heap.

We will now define some important terms related to symbolic objects:

Definition 5 (Symbolic Objects Extended)**.** *Lets us extend our definition of symbolic objects in Definition 1 (Section 3.2) to include reverse pointers as* $s : \langle \omega, \tau, \phi, \vartheta, \mu, \Upsilon_\psi, \Upsilon_h, S_\Psi \rangle$*, where* Υ_ψ *is the set of reverse stack pointers,* Υ_h *is the set of reverse heap pointers, and* S_Ψ *is the set of stacks that share the object.*

Definition 6 (Containers, $P^1 : s \times \Psi \rightarrow G$)**.** *The containers (or parents) of a symbolic object* s *in* Ψ *is a set of symbolic objects* $G \mid \forall e \in G\, \exists p \in s.\Upsilon_h (p.S = e \wedge \Psi \in e.S_\Psi)$.

The *containers* of a symbolic object s are all of the symbolic objects that contains s and belongs to the same execution environment Ψ as that of s. We could easily evaluate containers using the reverse heap pointer as specified in Definition 5.

Definition 7 (Transitive Closure of Containers, $P^+ : s \times \Psi \rightarrow G$)**.** *The transitive closure of containers of a symbolic object* s *in* Ψ *is a set of symbolic objects* G*, defined as:*

$$P^+(s,\Psi) = \begin{cases} \emptyset & \text{if } |P^1(s,\Psi)| = 0 \\ P^1(s,\Psi) \bigcup_{e \in P^1(s,\Psi)} P^+(e,\Psi) & \text{otherwise.} \end{cases}$$

Definition 8 (Reflexive Transitive Closure of Containers, $P^* : s \times \Psi \to G$)**.** *The reflexive transitive closure of containers of a symbolic object s in Ψ is a set of symbolic object G, defined as: $P^*(s,\Psi) = \{s\} \cup P^+(s,\Psi)$.*

Similarly, we can trivially evaluate immediate children of a symbolic object ($C^1 : s \times \Psi \to G$), transitive closure of children ($C^+ : s \times \Psi \to G$), and reflexive transitive closure of children ($C^* : s \times \Psi \to G$) using $s.\mu$. Using these definitions, let us discuss two algorithms that contribute to the lazy cloning and maximal sharing behavior.

Algorithm 1 Cloning an Execution Environment, $clone : \Psi \to \Psi$.

Require: Ψ_c, the current execution environment
Require: Ψ'_c, empty execution environment
1: **for** all stack frames $F_i \in \Psi_c$ **do**
2: $F'_i \leftarrow \langle F_i.C, A'_\mu, M'_\mu \rangle$ {A'_μ and M'_μ are shallow copy of $F_i.A_\mu$ and $F_i.M_\mu$, resp.}
3: push $F'_i \to \Psi'_c$
4: **for** all pairs $k \mapsto v \in F'_i.M_\mu$ **do**
5: {Mark the reachable objects of v as being shared by the new stack Ψ'_c}
6: **for** all $e \in C^*(v, \Psi_c)$ **do**
7: $e.S_\Psi \leftarrow e.S_\Psi \cup \{\Psi'_c\}$
8: $e.\Upsilon_\psi \leftarrow e.\Upsilon_\psi \cup \{\langle F'_i, k \rangle\}$
9: **end for**
10: **end for**
11: **end for**
12: **return** Ψ'_c

When CriticAL encounters a branch, it clones the current stack so that the cloned version can be used in the other branch (Algorithm 1). Note that only the stack is cloned at this point but symbolic objects remain shared between the two clones. When a symbolic object is mutated in one of the stacks, then we selectively clone all of the symbolic objects that are ancestors of the symbolic objects being mutated (Algorithm 2).

Algorithm 1 discusses cloning of the execution stack ($clone : \Psi \to \Psi$). Figure 3.6 illustrates the result of applying the algorithm on a sample code. The cloning starts at

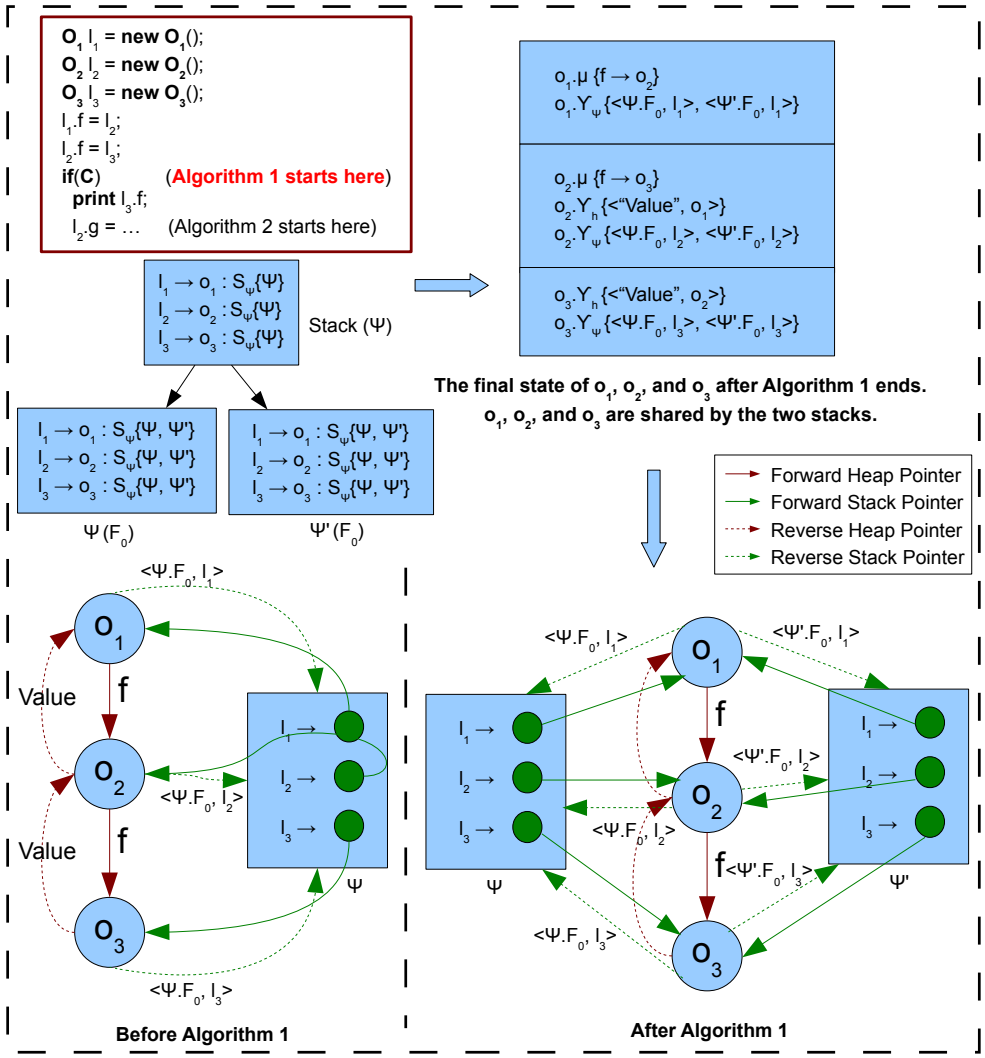


Figure 3.6: Illustration of Algorithms 1 on a sample code.

the `If` statement, which clones the current execution stack Ψ to produce Ψ'. Note that the stack frame of the two stacks only maintains a shallow copy of the maps (A_μ and M_μ), thus, sharing the same copy of the symbolic objects. The object heap of each symbolic object is then marked as being shared by the two stacks: Ψ and Ψ'.

In this way, even though the execution environment is cloned, the symbolic objects are not immediately cloned. When a symbolic object is about to be mutated, CriticAL will force the cloning of the object if it is shared by multiple stacks. Algorithm 2 communicates the major ideas behind the cloning algorithm for symbolic objects ($clone : S \times \Psi \rightarrow S$).

Figure 3.7 shows the algorithm in action reusing the same sample code of Figure 3.6. Note that Algorithm 2 does not start until the code mutates the object o_2. The algorithm first recursively reaches the top-most ancestors in the object graph of the symbolic object being cloned. It then starts the cloning from the top-most ancestor by updating the object's reverse stack and heap pointers. The process continues until the algorithm reaches the starting symbolic object. Note that even though o_1 and o_2 are cloned, o_3 is still shared. Hence, we achieve maximum sharing with the help of this lazy cloning strategy.

3.8 Non-Escaping Newly Created Objects

A stack frame besides $\langle C, A_\mu, M_\mu \rangle$ (see Definition 2) also maintains a set of newly created objects (using the `new` operator) during the execution of the corresponding method. Based on this set we can evaluate whether a newly created object escapes the method boundary. An object can escape the method boundary if it is assigned to a global variable (static field), if it is a part of the heap of the `this` object, or if it is a part of the heap of one of the parameters of the method. CriticAL allows us to enforce API-logic at various points of interest including the end of a method. Knowing non-escaping objects may help in identifying problems immediately at the end of a method, the information, which otherwise may be lost at the end of the path.

Let us extend Definition 2 as $F : \langle C, A_\mu, M_\mu, N \rangle$, where N is the set of newly created objects. Let $F_m \in \Psi$ be the stack frame for an instance method m, *this* be the symbolic object for the receiver and $p_1, \ldots, p_n$ be the symbolic objects for parameters of the method. Also let $S_F = \{v : f \mapsto v \in \Psi.G_\mu\}$ and $U_{sf} = \bigcup_{e \in S_F} C^*(e, \Psi)$ The non-escaping newly created objects set is defined as:

$$N^E_{NCO}(F_m) = \begin{cases} \Psi.F_m.N & m \equiv \text{entry} \\ \Psi.F_m.N - C^*(this, \Psi) - C^*(p_1, \Psi) - \ldots - C^*(p_n, \Psi) - U_{sf} & otherwise \end{cases} \tag{3.23}$$

Note that at the end of the method when the top stack frame F_{n-1} is popped, F_{n-2} updates

Algorithm 2 Cloning a Symbolic Object, $clone : \Psi \times S \rightarrow S$.

Require: Ψ_c, the current execution environment
Require: s, the symbolic object to be cloned
for all $o \in P^1(s, \Psi_c)$ **do**
  $clone(o, \Psi_c)$ {Recursively clone parents}
end for
{Assume that we maintain a list of processed objects to avoid cycles}
if $\Psi_c \notin s.S_\psi \vee |s.S_\psi| = 1$ **then**
  return s {Does not need to be cloned}
end if
$s' \equiv$ The shallow copy of s where $s'.\mu, s'.\Upsilon_\psi, s'.\Upsilon_h$, and $s'.S_\Psi$ are all new empty sets.
{Re-establish information about the stack for the clone and the original}
for all $p \in s.\Upsilon_\psi$ **do**
  if $p.F_* \in \Psi_c$ **then**
    $s'.\Upsilon_\psi \leftarrow \{\langle p.F_*, p.Value \rangle\}$
    $p.F_*[p.Value] = s'$
    $s'.S_\psi \leftarrow \{\Psi_c\}$
    $s.S_\psi \leftarrow s.S_\psi - \{\Psi_c\}$
    remove p from $s.\Upsilon_\psi$
  end if
end for
{Add the reverse heap pointers information in children}
for all $k \mapsto v \in s.\mu$ **do**
  $s'.\mu \leftarrow s'.\mu \cup \{k \mapsto v\}$
  $k.\Upsilon_h \leftarrow k.\Upsilon_h \cup \{\langle \text{“}Key\text{”}, s' \rangle\}$
  $v.\Upsilon_h \leftarrow v.\Upsilon_h \cup \{\langle \text{“}Value\text{”}, s' \rangle\}$
end for
{Remove extra heap pointers and update parents about the newly cloned child}
for all $h \in s.\Upsilon_h$ **do**
  if $\Psi_c \in h.S.S_\psi$ **then**
    remove h from $s.\Upsilon_h$
  end if
end for
for all $h \in s'.\Upsilon_h$ **do**
  if $\Psi_c \notin h.S.S_\psi$ **then**
    remove h from $s'.\Upsilon_h$
  else if $h.P_t =$ “Key” **then**
    $v = remove(h.S.\mu[s])$
    $h.S.\mu[s'] = v$
  else
    $\{h.P_t =$ “$Value$”$\}$
    for all $k \mapsto s \in h.S.\mu$ **do**
      $h.S.\mu[k] = s'$
    end for
  end if
end for
return s'

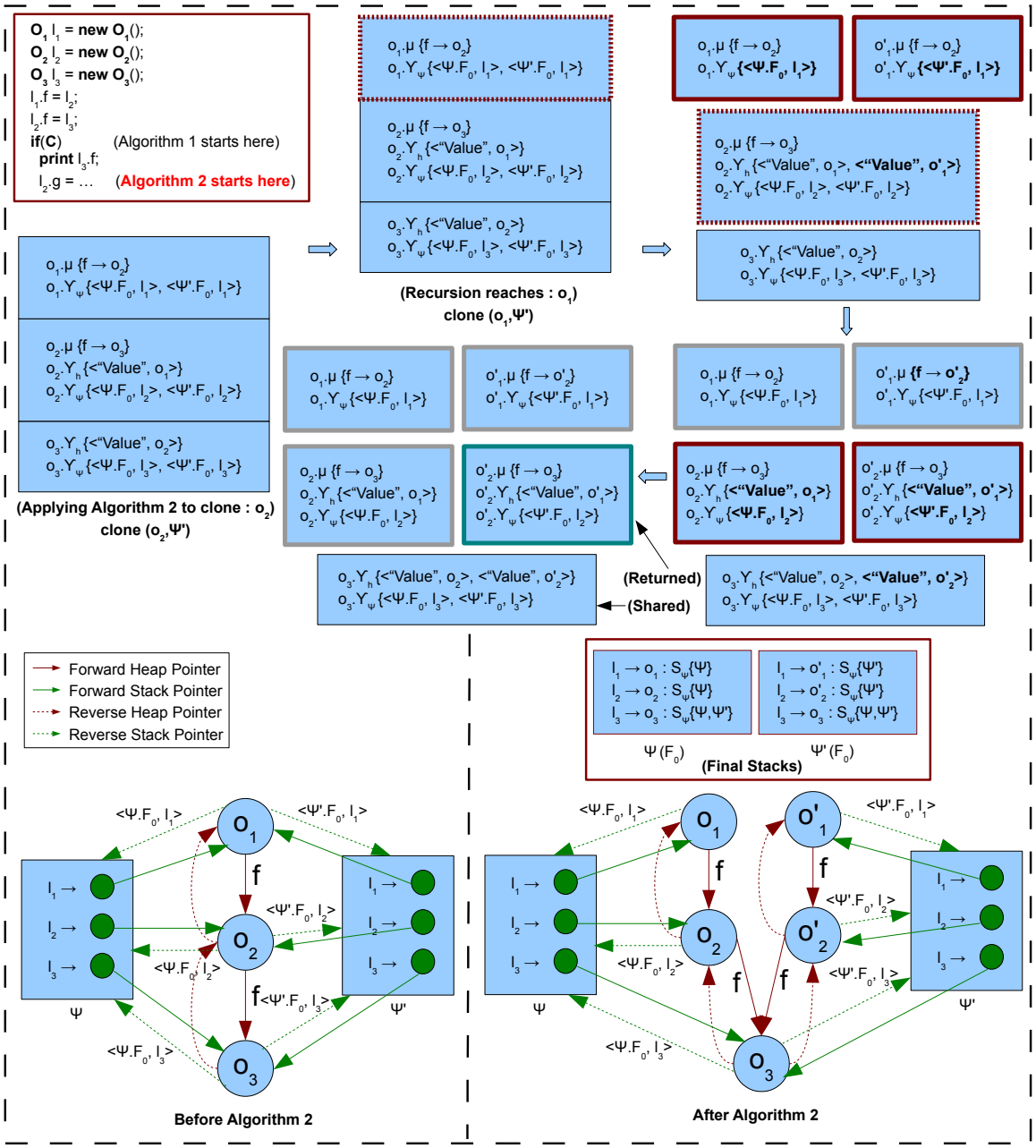


Figure 3.7: Stepwise illustration of running Algorithms 2 on a sample code.

its N as follows: $F_{n-2}.N \leftarrow F_{n-2}.N \cup F_{n-1}.N - N^{E}_{NCO}(F_{n-1})$.

3.9 Conclusion

In this chapter, we discussed the semantics of translating Jimple expressions to CriticAL expressions. We also discussed case-by-case rules representing the semantics of executing each Jimple statement. There were also discussions on how we could curb path explosion by setting a maximum bound on loop unrolling and had a glimpse at the future work of integrating a constraint solving module. We discussed in detail algorithms for maximizing sharing and minimizing cloning through a lazy initialization method. In the next chapter, we will present some implementation specific discussions on extending CriticAL to support APIs and libraries.

Chapter 4

Extending CriticAL

In Chapter 3, we covered the core design of CriticAL using abstract notation. In this chapter, we will discuss the implementation details behind extending CriticAL to support new APIs and libraries with the help of real examples and code snippets. Reconsider Figure 3.1 in Chapter 3 that explains the plugin mechanism of CriticAL. We will need to develop a plugin project in Eclipse and register the project to the CriticAL's core as an extension plugin. A new API can be supported by CriticAL by extending a set of base classes that represent symbolic objects. In the next few sections, we will discuss the details of implementing an extension plugin.

4.1 Overview of the Extension Process

Here we provide an overview of the process for developing an extension plugin for CriticAL:

1. Implement an extension plugin and specify the factory class (that implements the `IFactory` interface) for the extension plugin in `plugin.xml` . The registered factory class must specify entry methods from which a symbolic execution should start (Section 4.2). Entry points must be carefully selected to prevent a relevant API client code from being sliced away in a symbolic execution. The symbolic execution will start and end at an entry method and hence will ignore methods that are not reachable from

the method.

2. Implement necessary symbolic classes by extending the `SymbolicAPI` class provided by CriticAL to abstract the behavior of the corresponding Java API classes. Implement necessary methods within the extension classes to abstract the behavior of the corresponding Java API methods. Create a mapping between the symbolic types to Java types in the plugin factory class. CriticAL binds the Jimple parameters to corresponding symbolic object arguments. The extension plugin developer may modify the state of the receiver and parameters in the extension method. They could also perform a pre/post condition checking to generate a critique (which implements the `ICritic` interface) within this API method in the same way as pre/post-condition-based checking is done in program verification tools (Section 4.5).

3. If a critique needs to be generated outside the implementation of an API method, then the `ICheckPoint` interface can be used. Each `ICheckPoint` object represents a *Point-of-Interest (POI)* where a plugin developer specifies logic for checking the state of symbolic objects. For instance, we have a rule that checks that all `JFrame`s must eventually be visible at the end of the execution path. This rule cannot be implemented within an API method of the symbolic class for `JFrame`. Hence, we specify this logic in the `check()` method of the `PathEnd` POI. The implemented POI class then needs to be registered to the factory extension class. Several points of interest are provided for checking, e.g., `PathStart`, `PathEnd`, `MethodStart`, `MethodEnd`, `StatementStart`, `StatementEnd`, and so on. Note that the `MethodStart` and `MethodEnd` POIs are points before a method in the API-client code is expanded and after the statements within the method are executed, respectively. These POIs are different from API methods that are directly executed with the user provided semantics and checking logic. For each POI, the `check()` method provides all of the information about the program state, which includes the current statement, current stack frame, current execution stack, and the heap. Users can formulate any rules to check the pre/post condition or state invariants as they would do using other program verification techniques

(Section 4.4).

4. Finally, documents associated with critiques can be locally stored within the extension project in the "project/doc" directory or can be remotely stored in a web server and retrieved using HTTP.

4.2 Creating an Extension Plugin Project

Creating an extension plugin is a trivial task. Eclipse provides a wizard to create an extension plugin. After creating an empty extension plugin with a singleton option and a plugin activator, we will add new classes to the project to make it work with the core of CriticAL. Figure 4.1 shows the typical structure of a plugin extension project, the `plugin.xml` settings file, and the `IFactory` interface found in the `edu.clarkson.serl.critic.factory` package of the core (from here onward we will use the short form `serl` for `edu.clarkson.serl`). We have divided our extension plugin into three packages. In the subsequent sections, we will discuss the details of each package. Note that the full source code (LGPL) of the core and the swing extension can be downloaded from `https://sf.net/p/critical/code`.

4.3 The `serl.critic.swing` Package

There are two classes in this package. However, the `SwingCriticPlugin` is not very interesting; it just serves as a plugin activator and contains the auto-generated code from the plugin creation wizard. The `SwingFactory` class, however, is the interesting one, which interacts with the core to provide support for the Swing API. `SwingFactory` implements `IFactory` (Figure 4.1). This class is responsible to map the name of a class in `String` to the reflection `Class` API, which will be used by the core to call methods through Java's reflection support. Here is a high-level description of the methods:

- The `getSupportedTypes()` method returns a set of class names that are supported by the extension plugin, e.g., {`"javax.swing.JButton", "javax.swing.JPanel"`}.

Listing 4.1: Code for choosing entry points in the extension plugin.

```
1 public class SwingFactory implements IFactory {
2  Set<SootMethod> entryMethods;
3
4  public Set<SootMethod> getEntryMethods() {
5   return this.entryMethods;
6  }
7
8  public void checkEntry(SootMethod method) {
9   if(EntryFinder.isIntializedTypeOrSuperType(JFrameAbstraction.TYPE, method))
10    this.entryMethods.add(method);
11  }
12  //... elided
13 }
14
15 // EntryFinder is defined in the edu.clarkson.serl.critic.util package.
16 public class EntryFinder {
17  public static boolean isIntializedTypeOrSuperType(String qName, SootMethod m) {
18   RefType parentType = Scene.v().getRefType(qName);
19   SootClass parentClass = parentType.getSootClass();
20
21   JimpleBody jimpleBody = (JimpleBody)m.retrieveActiveBody();
22   for(Unit u : jimpleBody.getUnits()) {
23    if(u instanceof AssignStmt) {
24     Value rightOp = ((AssignStmt)u).getRightOp();
25     if(rightOp instanceof NewExpr) {
26      NewExpr newExpr = (NewExpr)rightOp;
27      RefType initializedType = newExpr.getBaseType();
28      SootClass initializedClass = ((RefType)initializedType).getSootClass();
29
30      if(initializedClass.equals(parentClass))
31       return true;
32
33      while(initializedClass.hasSuperclass()) {
34       initializedClass = initializedClass.getSuperclass();
35       if(initializedClass.equals(parentClass))
36        return true;
37      }
38     }
39    }
40   }
41   return false;
42  }
43  // ... elided
44 }
```

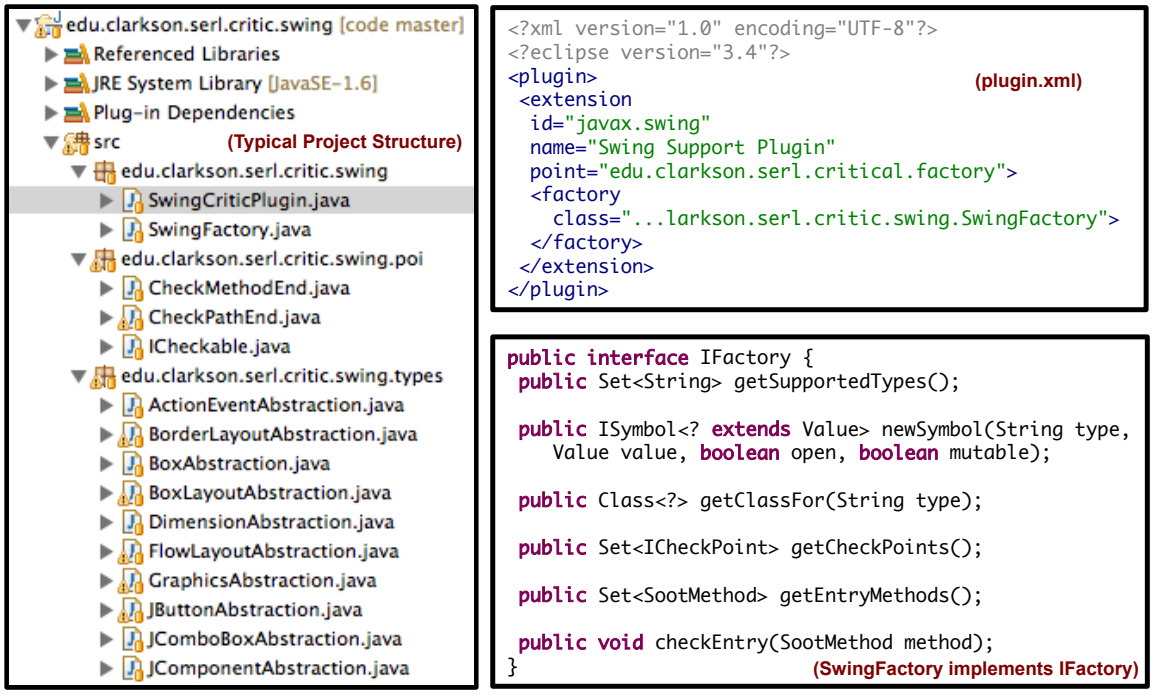


Figure 4.1: A typical structure of an extension plugin project, an XML settings file, and the `IFactory` interface.

- The `newSymbol()` method should create a new symbolic object given the fully qualified name of the Java type.

- The `getClassFor()` method maps a type name in `String` to the corresponding `Class` declared in the extension (see Section 4.5).

- The `getCheckPoints()` method returns the classes that represent the Points-of-Interest (POI) (see Section 4.4) where we will check the properties of the symbolic object to generate critiques.

- The `getEntryMethods()` method returns a set of methods from the application class (users' code) that serves as a starting point for symbolic execution.

- The `checkEntry()` method will essentially inform the extension plugin that the supplied method is a part of the client code. The extension plugin can evaluate its Jimple structure to determine if it wants to mark the method as an entry method to be later

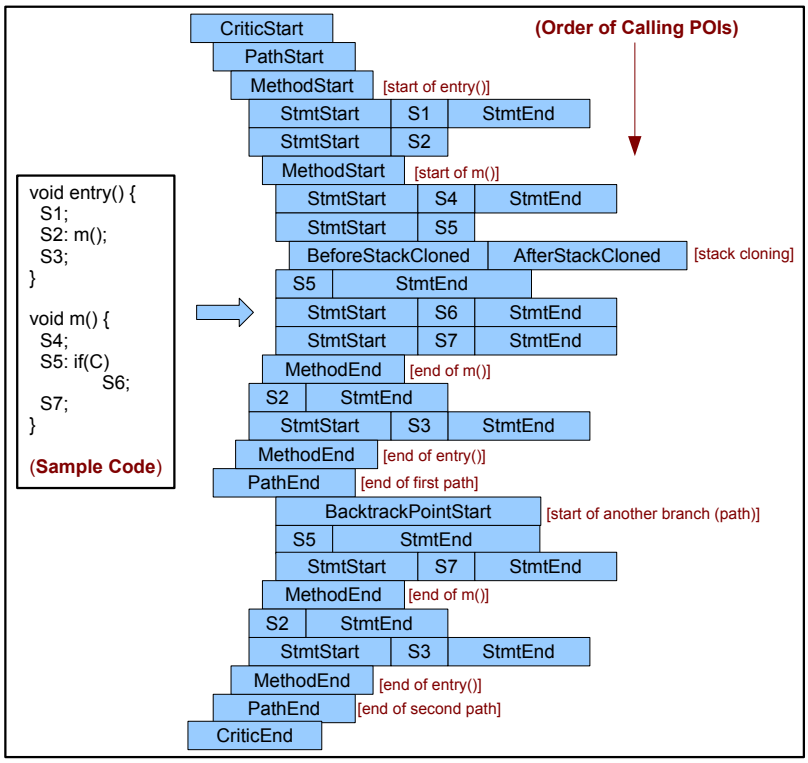


Figure 4.2: The order in which CriticAL executes the `check()` method of each POI.

returned by `getEntryMethods()`. For Swing, the creation of a GUI starts by initializing a top-level widget such as `JFrame`, which is followed by adding other widgets to the frame as well as event listeners. Hence, it makes sense to mark an API client method that intializes a `JFrame` as an entry method. An example is shown in Listing 4.1. The `checkEntry()` method calls a utility method that iterates through all of the Jimple statements in the supplied API client method looking for an assignment statement whose right hand side expression is a `new` expression, which intializes a `JFrame` or its subclass.

4.4 The `serl.critic.poi` Package

The `serl.critic.poi` package declares classes that represent POIs, the execution points where we would like to check the properties of symbolic objects. A POI class implements the `ICheckPoint` interface defined in the `serl.critic.extension` package of the core.

Listing 4.2: The `ICheckPoint` Interface.

```
1 public interface ICheckPoint {
2  public enum Interest {
3   CriticStart, CriticEnd, PathStart, PathEnd, MethodStart, MethodEnd,
4   StatementStart, StatementEnd,
5   BeforeStackCloned, AfterStackCloned, BackTrackPointStart
6  }
7  public Interest getInterest();
8  public IResult check(Stmt programCounter, IStackFrame frame, IStack stack,
9   Set<ISymbol<? extends Value>> heap);
10 }
11
12 public class CheckMethodEnd implements ICheckPoint {
13  public Interest getInterest() {
14   return ICheckPoint.Interest.MethodEnd;
15  }
16  public IResult check(Stmt programCounter, IStackFrame frame, IStack stack,
17    Set<ISymbol<? extends Value>> heap) {
18   Set<ISymbol<? extends Value>> set = frame.getNonEscapingNewObjects();
19   Result result = new Result(Interpreter.VOID);
20   for(ISymbol<? extends Value> s : set) {
21    if(s instanceof ICheckable) {
22     IResult tempResult = ((ICheckable<?>)s).checkAtEnd();
23     result.add(tempResult);
24    }
25   }
26   return result;
27  }
28 }
```

Listing 4.2 shows the `ICheckPoint` interface. The POIs (`ICheckPoint.Interest`) are self-explanatory and their order of execution is shown in Figure 4.2. We have used two POIs for Swing extension: `MethodEnd` and `PathEnd`, that are initialized in `SwingFactory` and returned by the `getCheckPoints()` method (see project structure in Figure 4.1. The `check()` method in a POI object provides all of the information needed to check the properties of a symbolic object with the state of the program at that point (program counter, most recent stack frame, stack, and heap represented as a set of symbolic objects). Note that all of the symbolic objects used in CriticAL are subtypes of the `ISymbol` interface, which is further discussed in Section 4.5.

Consider the `CheckMethodEnd` class, an implementation of `ICheckPoint`, in Listing 4.2.

We want to check properties of certain symbolic objects at the end of an API-client method, which is why we have this POI implemented. In particular, we want to check the properties of newly created objects that do not escape the method boundary. We get a set of such objects in line 18, the details of which have been covered in Rule 3.23 (Section 3.8). Using this set, we can enforce a rule such as: *a `JFrame` must eventually be visible once it is initialized.* Rather than cluttering the `check()` method with many such rules, we devised an interface called `ICheckable` that objects like `JFrame` could implement and specify such logic in the `checkAtEnd()` method of the interface in an elegant way. Also notice that the result of checking is encapsulated within an `IResult` object. This object encapsulates the result of executing an API-support method and all of the critiques associated with the method. Listing 4.3 shows both `IResult` and `ICritic` interfaces. Note that besides the specified POIs in the `ICheckPoint` interface, we could also check pre-conditions, post-conditions, and state invariants within the supported API-method, which is discussed in Section 4.5.

4.5 The `serl.critic.types` Package

In the `serl.critic.types` package, we declare all of the Swing classes that we would like to support. Figure 4.1 shows some of the classes that we have supported in our extension plugin. To support an API class, the extension plugin has to inherit the `SymbolicAPI` class defined in the core of CriticAL. Figure 4.3 shows the partial type hierarchy of the core, the swing extension plugin, and the Java Swing API. Let us discuss some of the implementation details of the `JFrame` shown in Listing 4.4.

We need to implement the `ISymbol` interface provided by CriticAL to support an API class such as `JFrame`. To make the implementation easy, CriticAL provides a subtype, the `SymbolicAPI` abstract class, that comes preloaded with all of the functionalities provided to a symbolic object. After creating an API class (`JFrameAbstraction`), the factory class of the extension plugin (`SwingFactory`) needs to map the qualified name of the real class to the symbolic version, i.e. `"java.swing.JFrame"` $\mapsto$ `JFrameAbstraction.class`, for the im-

Listing 4.3: `IResult` and `ICritic` Interfaces.

```
1 public interface IResult {
2  public ISymbol<? extends Value> getValue(); // Result of execution
3  public SortedSet<ICritic> getCritics();
4  public boolean add(ICritic critic);
5  public boolean remove(ICritic critic); ...
6 }
7
8 public interface ICritic extends Comparable<ICritic> {
9  public static final String LINE_NUMBER = IMarker.LINE_NUMBER;
10  public static final String URL = "url"; ...
11
12  // Three kinds of critiques
13  public static enum Type { Recommendation, Explanation, Criticism}
14  public static enum Priority {Low, Medium,High}
15
16  public String getId();
17  public String getOutermostClass();
18  public int getLineNumber();
19  public Type getType();
20  public Priority getPriority();
21  public String getTitle();
22  public String getDescription();
23  public Object getAttribute(String key);
24  public Object setAttribute(String key, Object value);
25  public Map<String, Object> getAttributeMap();
26 }
```

plementation in Listing 4.4. We implement this mapping in `SwingFactory.getClassFor()` as shown in Listing 4.5. After this association has been established, CriticAL can then delegate all of the operations on a real Java object to the associated symbolic object, thus, symbolically executing the API client code.

Listing 4.6 shows all of the fields (properties) for `JFrame` and `JComponent` that we have modeled in the current implementation of the Swing plugin. An API method will update these properties as its execution semantics. We saw an example in Listing 4.4 where the `setDefaultCloseOperation()` API method mapped the field, `DCO` to the supplied symbolic value in the parameter. The state of a symbolic object is represented by the association of these symbolic properties to symbolic objects at a given control point. Note that all of the GUI widgets in Swing inherit from the `JComponent` class, hence, our abstraction classes

Listing 4.4: Implementation of the symbolic `JFrame` class.

```
1  public class JFrameAbstraction extends SymbolicApi<Value>
2                                  implements ICheckable<Value> {
3   public static final SymbolicKey DCO = SymbolicKey.fromObject("dco");
4
5   public IResult execute(InvokeExpr invkExpr, List<ISymbol<? extends Value>> args) {
6    String method = invkExpr.getMethod().getName();
7    if(method.equals("<init>")) // Implementation of JFrame's constructor
8     return this.init(args);
9    if(method.equals("setDefaultCloseOperation"))
10    return this.setDefaultCloseOperation(arguments);
11   ...
12  }
13
14  public IResult init(List<ISymbol<? extends Value>> arguments) {
15   this.put(DCO, ConstInteger.fromInteger(JFrame.HIDE_ON_CLOSE)); ...
16   return new Result(Interpreter.VOID);
17  }
18
19  public IResult setDefaultCloseOperation(List<ISymbol<? extends Value>> arguments) {
20   ISymbol<? extends Value> op = arguments.get(0);
21   this.put(DCO, op);
22   return new Result(Interpreter.VOID);
23  }
24
25  public IResult checkAtEnd() {
26   Result result = new Result(Interpreter.VOID);
27   ...
28   ISymbol<? extends Value> closeOp = this.get(DCO);
29   int value = (Integer)closeOp.getValue();
30   ICritic critic = new Critic(
31     ...
32     ICritic.Type.Recommendation,
33     ICritic.Priority.High,
34     "You are using the " + toCloseOperationString(value) + " property for the default ..." +
35     "There are multiple options available for closing a frame. Please click to see the details."
36     );
37   critic.setAttribute(ICritic.URL, "http://docs.oracle.com/..."); // Location of document
38   result.add(critic);
39   return result;
40  }
41
42  public static String toCloseOperationString(int value) {
43   if(value == JFrame.HIDE_ON_CLOSE)
44    return "HIDE_ON_CLOSE";
45   if(value == JFrame.EXIT_ON_CLOSE)
46    return "EXIT_ON_CLOSE";
47   ...
48  }
49  ...
50 }
```

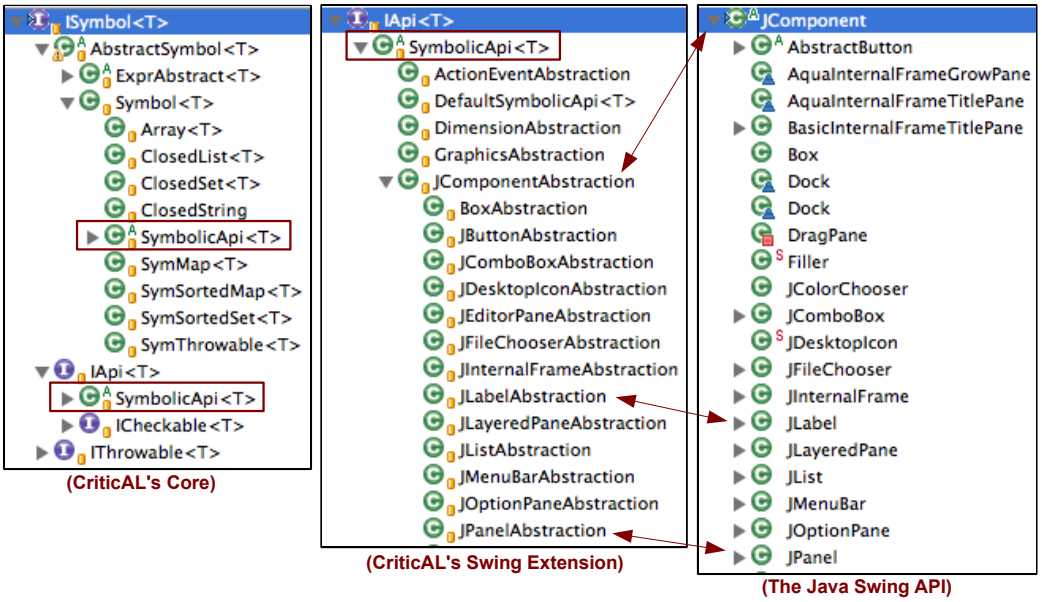


Figure 4.3: The type hierarchy of the core, the Swing extension plugin, and the Swing API.

Listing 4.5: Implementation of `SwingFactory`.

```
1 public class SwingFactory implements IFactory {
2  public Class<?> getClassFor(String type) { ...
3   if(type.equals("java.swing.JButton"))
4    return JButtonAbstraction.class;
5   if(type.equals("javax.swing.JFrame"))
6    return JFrameAbstraction.class; ...
7  } ...
8 }
```

for the GUI widgets also inherit from the `JComponentAbstraction` class and reuse these fields in defining the abstractions for their API methods. However, these fields only model a subset of the properties of `JFrame` and `JComponent` that is good enough for us to check the GUI composition and layout logic.

To illustrate the symbolic execution process, reconsider Figure 1.5 (Section 1.1.2). The statement in line 15 (Figure 1.5.(a)), first calls `SwingFactory.newSymbol("java.swing.JFrame", ...)`, which creates a new `JFrameAbstraction` object and returns to the core. The core maps the local variable `frame` in line 15 to the newly created symbolic `JFrame` in the stack. After this operation, the core encounters a call to the method representing the

Listing 4.6: Fields that model the state of `JFrame` and `JComponent`.

```
1 public class JFrameAbstraction extends SymbolicApi<Value> implements ICheckable {
2  public static final SymbolicKey TITLE = SymbolicKey.fromObject("title");
3  public static final SymbolicKey CONTENT_PANE = SymbolicKey.fromObject("cPane");
4  public static final SymbolicKey DCO = SymbolicKey.fromObject("DCO");
5  public static final SymbolicKey LAID_OUT = SymbolicKey.fromObject("laidout");
6  public static final SymbolicKey VISIBLE = SymbolicKey.fromObject("visible");
7  public static final SymbolicKey MENU_BAR = SymbolicKey.fromObject("menu");
8  ...
9 }
10
11 public class JComponentAbstraction extends SymbolicApi<Value> implements ... {
12  public static final SymbolicKey LOCATION = SymbolicKey.fromObject("location");
13  public static final SymbolicKey SIZE = SymbolicKey.fromObject("size");
14  public static final SymbolicKey PREFERRED_SIZE = SymbolicKey.fromObject("pSize");
15  public static final SymbolicKey MIN_SIZE = SymbolicKey.fromObject("minSize");
16  public static final SymbolicKey MAX_SIZE = SymbolicKey.fromObject("maxSize");
17
18  public static final SymbolicKey LAYOUT = SymbolicKey.fromObject("layout");
19  public static final SymbolicKey COMPONENTS = SymbolicKey.fromObject("comps");
20  public static final SymbolicKey VALID = SymbolicKey.fromObject("valid");
21  public static final SymbolicKey PARENT = SymbolicKey.fromObject("parent");
22  ...
23 }
```

constructor of the frame, which is delegated to the `JFrameAbstraction.init()` method (lines 14-17) through the `JFrameAbstraction.execute()` method (lines 5-12) in Listing 4.4. All of the API class needs to implement the `IResult execute(InvokeExpr invkExpr, List<ISymbol<? extends Value>> args)` method to receive a method call on the symbolic counterpart. Also notice how the `execute()` method of `JFrameAbstraction` switches between different API method calls based on the name. Hence, the support for method overloading must be implemented explicitly by the supported API class in the `execute` method.

In our implementation, we are supporting the recommendation for the default closing behavior of `JFrame`. In the constructor (lines 14-16), we specify that the default closing behavior for a newly created `JFrame` is `HIDE_ON_CLOSE`. Notice that we are implementing the `ICheckable` interface to specify the checking logic at the `MethodEnd` POI (Section 4.4). Hence, when the symbolic execution reaches the end of a method that creates this sym-

bolic `JFrame` object, `checkAtEnd()` (lines 25-40) gets called. This method generates the recommendation shown in Figure 1.5(b). In summary, to generate a critique (explanation, recommendation, or criticism), we implement rules that check the state (property-value map) of a symbolic object at various points of interest. The final result is encapsulated in an `IResult` object and returned to the core. The core then automatically creates the necessary markers to display the critiques to the user in the Eclipse IDE.

4.6 Supporting Listeners

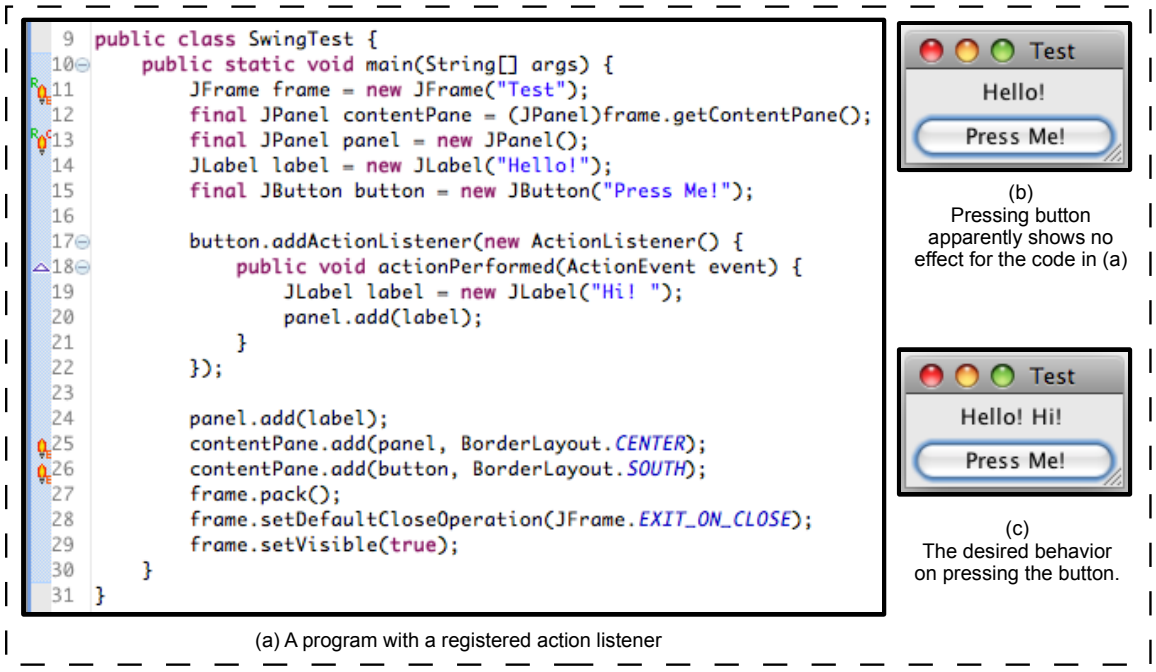


Figure 4.4: Code showing the use of an action listener in a `JButton`.

We have briefly discussed the support for listeners in Section 3.1 previously. Let us discuss in more details about listeners here. Listeners are treated as any other methods and in-lined at the end of each execution path if they are registered in the GUI objects in the path. Hence, a GUI object that accepts a listener needs to inform CriticAL that it has accepted the listener in the current path. This information is passed to the execution stack through the `ICallbackPoint` object, which will be discussed shortly. But for now,

let us look at the example shown in Figure 4.4. The code in the example creates a window shown in Figure 4.4(b). An action listener is registered to the `JButton`, in lines 17-22 (Figure 4.4(a)). The desired behavior of the button is shown in Figure 4.4(c) where the action listener adds the new label `"Hi!"` to the panel. However, when the user clicks the button, he does not get this behavior. Also note that CriticAL produces a criticism in line 13 of the code that informs the user about this problem with the panel, which is: *When the content of a container is changed, it must be revalidated and repainted for the change to take effect.* (see Section 2.3.4 for details about this rule). The problem can be fixed by adding `panel.revalidate(); panel.repaint()` just after line 20 in Figure 4.4(a). Let us now discuss how CriticAL produces this criticism.

Listing 4.7 shows the `ICallbackPoint` interface (lines 1-10) that is used to represent a listener object. For the type of the listener object in Figure 4.4 we would use `"java.awt.event.ActionListener"`. Whenever the `addActionListener()` method is called on a `JButton`, lines 19-27 of Listing 4.7 gets executed. There is only one method in the action listener, i.e., `actionPerformed()` that needs to be supported. Lines 21-27 dynamically prepares arguments for this method to be used by the core for binding parameters to arguments. Notice how `ActionEvent` (Figure 4.4(a), line 18) is mapped to `ActionEventAbstraction` in Listing 4.7 (lines 23-25). After we have configured a listener, the code inside the listener methods get executed in the same way as a regular method is executed. Now, we just need to implement the necessary rule to detect the problem with changed containers as if it would occur in a regular method. This rule is specified in the `JComponentAbstraction` class, a super class of `JButtonAbstraction` as shown in Listing 4.8.

In summary, when the `panel.add(label)` get called in line 20 of Figure 4.4(a), the `JComponentAbstraction.add()` method in lines 5-9 of Listing 4.8 gets called. When the control reaches the end of the path, it evaluates the set of non-escaping newly created objects for the entry method and finds the `panel` object in the set. Hence, it executes the `checkAtEnd()` method (lines 11-32 of Listing 4.8) to report the criticism at line 13 of Figure 4.4(a). In this way, using the call back mechanism, we can easily provide support

Listing 4.7: The `ICallbackPoint` Interface and the symbolic `JButton` class.

```
1  public interface ICallbackPoint {
2   // The listener object
3   public ISymbol<? extends Value> getCallbackSymbol();
4   // The source object
5   public ISymbol<? extends Value> getSource();
6   // The fully qualified type of the listener
7   public String getCallbackType();
8   // To dynamically configure arguments for each listener method declared in the listener's type
9   public List<ISymbol<? extends Value>> getArguments(SootMethod method);
10 }
11
12 public class JButtonAbstraction extends JComponentAbstraction {
13  public IResult execute(InvokeExpr invkExpr, List<ISymbol<? extends Value>> args) {
14   String method = invkExpr.getMethod().getName();
15   if(method.equals("addActionListener")) {
16    return this.addActionListener(args);
17   }
18   // Do whatever JComponentAbstraction does with other methods
19   return super.execute(invokeExpr, args);
20  }
21
22  public IResult addActionListener(List<ISymbol<? extends Value>> args) {
23   // Create an ICallbackPoint object and dynamically configure the ActionEvent parameter
24   // for the "void actionPerformed(ActionEvent)" listener method
25   ICallbackPoint callBack = new CallbackPoint(..., "java.awt.event.ActionListener") {
26    public List<ISymbol<? extends Value>> getArguments(SootMethod method) {
27     ArrayList<ISymbol<? extends Value>> arguments = new ArrayList ...;
28     ActionEventAbstraction event = new ActionEventAbstraction();
29     event.put(ActionEventAbstraction.SOURCE, JButtonAbstraction.this);
30     arguments.add(event);
31     return arguments;
32    }
33   };
34
35   IStack stack = Interpreter.instance().peekStack();
36   stack.addCallback(callBack); // Register the callback object to the execution stack
37   return new Result(Interpreter.VOID);
38  }
39  ...
40 }
```

for listeners.

4.7 Action-Based Critiquing

We saw two instances of state-based critiquing (Listings 4.4 and 4.8) where the state of a symbolic object was used to critique an API-client code at the `MethodEnd` POI. Sometimes, actions are also necessary. For instance, the recommendation generated for the confusing APIs (`setAlignmentX()`, `setHorizontalAlignment()`, and `setHorizontalTextPosition()`, also discussed in Sections 1.1.1 and 2.5.2) use action rather than state. Listing 4.9 shows an implementation where the call to one of the confusing API methods generates recommendation for the use of other confusing API methods in the group and an explanation of the proper usage scenario for each method.

4.8 Supporting Static API Methods

Supporting a static method is pretty trivial. All one has to do is to map the name of the real Java class declaring the static method to the symbolic counter part in the extension factory and implement the static method with the same name. CriticAL will automatically delegate the call to the static API method to the corresponding static method in the symbolic API class. For instance, the `javax.swing.Box` class declares a bunch of static methods for creating different kinds of `Box`es such as `"static Component Box.createGlue()"`. Listing 4.10 shows an example of the corresponding static method in the `BoxAbstraction` class.

4.9 A Generalizability Argument

CriticAL works on top of Jimple IR and should be able to support APIs and libraries in any programming languages that could be transformed to Jimple IR. In this chapter, we explored the extension process of CriticAL to support the Swing API. We followed a general process described in Section 4.1 to support the Swing API. The same process could be applied to support other APIs as well, which serves as a general template for extending

CriticAL. In this section, we sketch the extension process for a different API (the `java.io` API) as an example.

Suppose we want to support two rules for the subclasses of `InputStream` and `OutputStream`:

1. A closed IO stream cannot be read or written again.

2. An opened IO stream must eventually be closed in the program.

Let us outline the process of extending CriticAL to support these rules using the general template discussed in Section 4.1:

1. Create an extension project for the IO API. Create a factory class by implementing the `IFactory` interface to associate the Java classes with the CriticAL counterparts that need to be supported. These classes would be `BufferedInputStream`, `FilterInputStream`, `BufferedOutputStream`, `FilterOutputStream`, and so on. Define entry points in the `checkEntry()` method of the factory class from which a symbolic execution should start. For the Swing extension we chose the API-client methods that initialized a top-level GUI (`JFrame`). For the IO API, a good heuristic would be the API-client methods that initialize one of the subclasses of `InputStream` and `OutputStream`. Note that using such a heuristic has a chance that a relevant API-client method is sliced away because it is unreachable in the call graph of the entry method, thus, making analysis unsound. The worst-case scenario of choosing an entry method would be the main method that would make the analysis sound but may produce many unnecessary execution paths. Hence, choosing an entry method is a compromise between performance and precision for an extension plugin developer.

2. Create necessary abstraction classes for the API classes in the IO API. At the top-level of the abstraction class hierarchy would be `InputStreamAbstraction` and `OutputStreamAbstraction` that would extend from the `SymbolicAPI` class present in the core of CriticAL. These classes would be inherited by subclasses such as `FilterInputStreamAbstraction`, `BufferedOutputStreamAbstraction`, and `FilterOutputStreamAbstraction`. The constructor of the `InputStreamAbstraction` and `OutputStreamAbstr-`

`action` will associate a symbolic property `CLOSED` with `false`. The `close()` methods of these classes will associate `CLOSED` with `true`. Any methods that abstract read and write operations in the abstraction classes will check if `CLOSED` is associated to `false`. If not so, then a criticism would be produced as a semantics of executing those API methods. In this way, the first rule could be supported.

3. To support the second rule, we need to implement a POI. We will implement the `MethodEnd` POI as discussed in Section 4.4 (Listing 4.2). We will then implement the `checkAtEnd()` method in `InputStreamAbstraction` and `OutputStreamAbstraction` that would check if `CLOSED` is associated to `true` (see Listing 4.4 for an example). If not so, we will generate a criticism saying the IO stream is not closed in the program.

Note that CriticAL has more features then the IO API would need, for example, the support for a callback method (an event listener method) that we did not use in this example. In summary, we see a broader prospect for CriticAL in the API-based programming because it appears to us that CriticAL can support more than just Swing and the Java IO APIs. Having implemented the support for one complex API (Swing) and sketched out the implementation for another API (java.io), we feel confident about the generalizability of CriticAL. Nevertheless, we need to test more APIs and libraries as a future work to convincingly make such a claim.

Listing 4.8: The `JComponentAbstraction` Class.

```
1  public class JComponentAbstraction extends SymbolicApi<Value>
2                                          implements ICheckable<Value> {
3   public static final SymbolicKey VALID = SymbolicKey.fromObject("valid");
4   public static final SymbolicKey VISIBLE = SymbolicKey.fromObject("visible");
5
6   public IResult add(List<ISymbol<? extends Value>> arguments) {
7    // Whenever a component is added, the container is in invalid state
8    this.put(VALID, Interpreter.FALSE);
9    ...
10  }
11
12  public IResult setVisible(List<ISymbol<? extends Value>> arguments) {
13   ISymbol<? extends Value> visibility = arguments.get(0);
14   this.put(VISIBLE, visibility);
15   for(ISymbol<? extends Value> child : this.getAllChildren()) {
16    child.put(VISIBLE, visibility);
17   }
18   ...
19  }
20
21  public IResult checkAtEnd() {
22   ...
23   // If a local component is visible but invalid
24   ISymbol<? extends Value> visible = this.get(VISIBLE);
25   if(visible != null && visible.equals(Interpreter.TRUE)) {
26    ISymbol<? extends Value> validity = this.get(JComponentAbstraction.VALID);
27    if(validity == null || validity.equals(Interpreter.FALSE)) {
28     ICritic critic = new Critic(
29        ...
30        ICritic.Type.Criticism,
31        ICritic.Priority.High,
32        "You have created a GUI widget ... that has been changed after it " +
33        "has been made visible. The effect of such change is not visible until " +
34        "widget.revalidate() followed by widget.repaint() is called or the " +
35        "pack() method of the JFrame/JDialog that contains it is called. "
36        );
37     critic.setAttribute(ICritic.URL, "http://docs.oracle.com/.../tutorial/.../problems.html");
38     result.add(critic);
39    }
40   }
41   return result;
42  }
43  ...
44 }
```

Listing 4.9: Action-based recommendation for confusing APIs.

```
1 public class JComponentAbstraction extends SymbolicApi<Value> implements ... {
2  public IResult execute(InvokeExpr invkExpr, List<ISymbol<? extends Value>> args) {
3   String m = invkExpr.getMethod().getName();
4   if(...) ...
5   else if(m.equals("setAlignmentX") || m.equals("setAlignmentY") ||
6     m.equals("setHorizontalAlignment") || m.equals("setVerticalAlignment") ||
7     m.equals("setHorizontalTextPosition") || m.equals("setVerticalTextPoisition")) {
8    if(this instanceof JButtonAbstraction || this instanceof JLabelAbstraction ||
9     this instanceof JRadioButtonAbstraction || this instanceof JComboBoxAbstraction) {
10    ICritic critic = new Critic(
11      ...
12      ICritic.Type.Recommendation,
13      ICritic.Priority.Medium,
14      "You are using an API that controls alignment for a GUI widget. " +
15      "setAlignmentX() and setAlignmentY() is designed to be used with BoxLayout ..."
16    );
17    // This time we are using a local document within the plugin project in "./doc" folder
18    String url = CriticPlugin.getDocumentURL(PLUGIN_ID, "RE-Alignment.html");
19    critic.setAttribute(ICritic.URL, url);
20    IResult result = new Result(Interpreter.VOID);
21    result.add(critic);
22    return result;
23   }
24  } ...
25 } ...
26 }
```

Listing 4.10: Support for a static method in the symbolic `Box` class.

```
1 public class BoxAbstraction extends JComponentAbstraction {
2  public static IResult createGlue(List<ISymbol<? extends Value>> arguments) {
3   BoxAbstraction box = new BoxAbstraction();
4   // The arguments has no effect so lets reuse it
5   box.init(arguments);
6   return new Result(box);
7  }
8  public IResult init(List<ISymbol<? extends Value>> arguments) {
9   IResult result = super.init(arguments);
10  BoxLayoutAbstraction layout = new BoxLayoutAbstraction();
11  layout.setDefault(true);
12  this.put(LAYOUT, layout);
13  return result;
14 }
15 ...
16 }
```

Chapter 5

Evaluation

CriticAL has been iteratively developed by adding new rules to its Swing extension plugin, fixing major design issues in supporting those rules, and testing against the unit test cases developed for each rule and against the real user programs over a period of a year. In this chapter, we will present two kinds of evaluations of CriticAL on real programs written by novice Swing users:

1. Formative study conducted on undergraduate students from Clarkson University.
2. Evaluation of programs on the Java Swing forum and official Swing tutorials.

5.1 Formative Study

When we developed the first prototype of CriticAL in Fall 2011, we only had a few rules implemented based on our experience with the Java Swing API. We did not have the case study of Chapter 2 conducted then. Hence, to get an initial assessment of its utility on the field and to identify room for improvement, we conducted a formative (or observational) user study [28] with a dozen undergraduate students in the *Software Design for Visual Environments (EE408, Fall 2011)* course offered by Dr. Daqing Hou at Clarkson University. The study was conducted on *Wednesday, October 19, 2011* and its main goal was to get a sense of the utility of the tool on real novice programmers and observe whether the tool

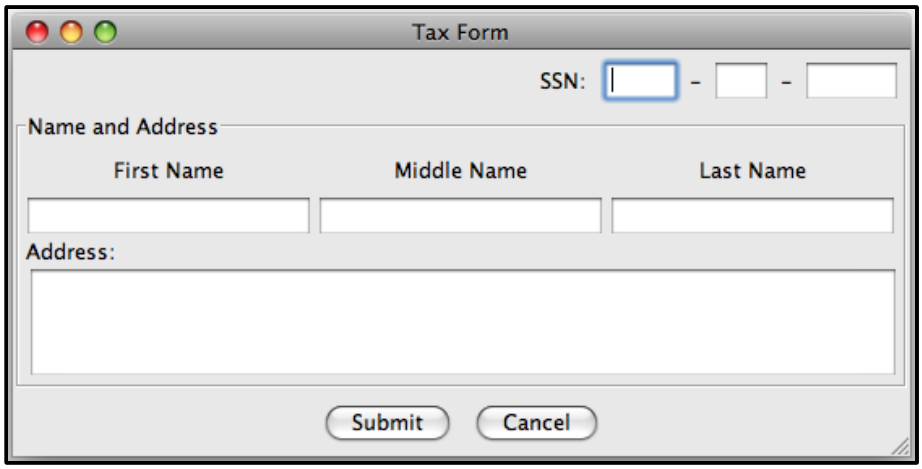


Figure 5.1: Design of a tax form.

could help with learning to use the Swing API. In this section, we report on the details of the study and our observations.

5.1.1 Methodology

We wanted to give a realistic problem to students in the study that involved components for which CriticAL had implemented some support. We asked them to design a tax form in the classroom similar to the one shown in Figure 5.1. The assignment required them to compose multiple GUI widgets using a combination of layout managers. They were allowed to use the Internet for help as well as CriticAL. All of their machines were equipped with Eclipse IDE with an installation of CriticAL. They were taught how to use CriticAL and view the critiques shown by the tool. They were given 45 minutes to solve the problem and were encouraged to use CriticAL whenever they encounter a problem. We asked them to inform us of any interesting observation (good or bad) about using CriticAL during the course of their programming by raising their hand. We took a note of their observations. We also observed the students ourselves when they were coding the solution by periodically walking around the room [1]. At the end of the class, we conducted a short one minute interview with each student and asked them about their experience with CriticAL. Note

[1] We did not use any audio or visual medium to record this experiment.

that we did not use any questionnaires in this study because we only wanted to observe the use of CriticAL as a formative (not summative) study and test whether there is some utility of the tool at its early prototype phase. We have collected a two A4 sheet size of notes from the study which contains a short description of problems that the students faced and their suggestions for the improvement of CriticAL.

5.1.2 Subjects

There were 12 senior undergraduate students who participated in this study from different majors: computer science, software engineering, and electrical engineering. Students were only taught the basic design principle of Swing and about some of the GUI widgets that they could use in the study. They also had one lecture worth of ideas about layout managers but did not have programming experience with combining multiple layout managers. In summary, they could be classified as novice users of the Java Swing API. Furthermore, they also knew that CriticAL was the tool developed by us and they were being asked to use it for a programming task.

5.1.3 Observation and Results

Almost all of the students referred to the online Swing tutorial for help with using layout managers in the beginning of the experiment. As they started writing code, they got stuck a couple of times and then they started using CriticAL. Once they found that CriticAL could provide them with useful suggestions, they used CriticAL more frequently. Here are some problems that CriticAL helped solve in the study:

Parent Switching (3 students)

We have previously discussed the parent switching behavior in Section 2.3.3. Surprisingly, all three students stumbled upon this problem for the content pane of the `JFrame`. The content pane of the `JFrame`, by default, is managed by `BorderLayout` and calling `contentPane.add(widget)` assigns the widget to the `CENTER` location of the layout. All of the three students

wanted to add multiple widgets to the content pane and called `contentPane.add(widget);` more than once, thus, overwriting the `CENTER` location more than once. As a result, only the last widget was visible and they were surprised at this behavior of the content pane. They did not understand that it was the layout manager that was responsible for this behavior. They liked it when CriticAL pointed out this problem to them.

Issues with `JFrame` (2 students)

One of the student forgot to call `JFrame.setVisible(true)` after laying out the content pane in his `JFrame` and wondered why his program immediately terminated after running without showing the window. CriticAL suggested that the `JFrame` was not set visible in his program after which the student fixed his code accordingly.

Another student noticed that even after closing his `JFrame` with the window manipulation button, the `JFrame` did not dispose properly. Even though it was invisible on the desktop, it was still running on the background as seen on the task bar. CriticAL's recommendation on considering multiple available options for the close operation helped clarify his situation. He added `JFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` to fix his problem.

Content Mismatch (1 student)

We have previously discussed the issue with content mismatch in Section 2.3.5. One of the students came up with a code snippet similar to Listing 5.1. The `addLayoutComponent()` method only added the three panels to the layout manager but not to `mainPanel`. Hence, the content of `mainPanel` and its layout manager `layout` was different. As a result, neither of the three panels were visible and the students was wondering what went wrong. CriticAL pointed out this issue to the student and presented a document that described the process of adding new widgets to a container.

Listing 5.1: Content mismatch between a `JPanel` and its layout manager.

```
1 JPanel mainPanel = new JPanel();
2 BorderLayout layout = new BorderLayout();
3 mainPanel.setLayout(layout);
4
5 JPanel ssnPanel = new JPanel();
6 JPanel namePanel = new JPanel();
7 JPanel buttonPanel = new JPanel();
8
9 // Only adds to the layout but not to mainPanel
10 layout.addLayoutComponent(ssnPanel, BorderLayout.NORTH);
11 layout.addLayoutComponent(namePanel, BorderLayout.CENTER);
12 layout.addLayoutComponent(buttonPanel, BorderLayout.SOUTH);
13 ...
```

Table Design (1 student)

We have previously discussed the issue with table design in Section 2.3.6. One of the students tried to achieve the table-like design (Figure 5.1) by arranging the three `JLabel`s for first, middle, and last names and the corresponding `JTextField`s using two different `JPanel`s. In the first `JPanel`, he set the layout manager to `BoxLayout` with horizontal alignment and added the three `JLabel`s. In the second `JPanel`, he used another `BoxLayout` with horizontal alignment and then added the three `JTextField`s. Both of the `JPanel`s were then added to another `JPanel` managed by another `BoxLayout` with vertical alignment. Nevertheless, the labels and text fields did not align properly to give a table-like view of Figure 5.1. CriticAL explained this problem to the student and suggested to choose from one of the following layout managers: `GridLayout`, `GridbagLayout`, and `SpringLayout`. Note that even though only one student was helped by this rule in the classroom during the study, several students reported to us that CriticAL helped them with the table-design while working on the problem as a homework.

5.1.4 Lessons Learned

Table 5.1 summarizes the result of our formative study. Even at its early stage of development, CriticAL helped 7 out of 12 students with the programming process. Nevertheless,

Table 5.1: Summary of the formative user study conducted on Clarkson's students.

Problems Helped by CriticAL	**Students #**
Parent Switching	3 (25%)
Issues with `JFrame`	2 (16.66%)
Content Mismatch	1 (8.33%)
Table Design	1 (8.33%)
Total Helped	**7 (58.33%)**
Total Students	**12**

due to some internal bugs, CriticAL crashed on 3 instances, which has been fixed now. Out of the 12 students, 4 of them were able to complete the task within the allocated time and the rest submitted the solution as a homework. We should not claim that the students who completed the task on time did so only because of CriticAL, however, our observation shows that CriticAL certainly provided them with some assistance. Furthermore, background knowledge and the ability to learn fast may also have played role in the success of the four students. In the informal one minute interviews at the end of the study, students suggested a few improvements for CriticAL such as implementing markers rather than just a list view for critiques, some refinement on the available documents by making them more elaborate, and support for more widgets than was available then. We have used those suggestions to improve CriticAL. In general, all of the students showed a positive response about their overall experience with CriticAL. Nevertheless, we should also note that the students had a prior knowledge that the tool was developed by us and may have given a biased opinion about their experience.

The study shows that novice programmers encounter problems when assigned with a new task. Having the Internet and Swing tutorials may not be sufficient for them to achieve a complex task in a limited time. CriticAL was helpful in spotting important problems at several instances and provided that extra help they needed, which otherwise would require an expert eye. To summarize, CriticAL does have some utility in the API-based programming practice. Note that even though this formative study was encouraging, we feel that a more thorough summative user study is needed to assess the overall quality

of critiques as the tool has been redesigned since this study.

5.2 Evaluation of CriticAL on Users' Programs

In this section we will evaluate CriticAL both for performance and utility using real users' programs. For performance, we will use 90 runnable programs collected in the case study of Chapter 2 from the Java Swing forum as well as 75 runnable programs downloaded from the official Swing tutorials [2]. For utility, we will use the samples collected from the forum only because we are familiar with the problems of the users and their source code, which will help us assess the utility of critiques more thoroughly. Furthermore, code from the official Swing tutorials are prepared by experts and would not have issues faced by real novice programmers.

5.2.1 Evaluation of Performance

We downloaded a fresh copy of the Eclipse Classic (version 3.7.1) IDE from [3]. After installing the latest version of CriticAL (version 1.0.20.beta) from [4], we configured two Java projects in the IDE: one for the code collected from the Swing forum (SF) and another for the code collected from the Swing tutorials (ST). Note that each project contained multiple executable programs: 90 in SF and 75 in ST. Both projects were run separately and were monitored using a specialized logging module developed just for the task.

The machine used for the test was a Mac Book Pro laptop with the following specification: 2.4 Ghz Intel Core 2 Duo Processor, 2 GB main memory, Mac OSX 10.5.8 operating system. The Eclipse IDE was configured with 800 MB of starting memory and 1024 MB of the maximum memory using Java Virtual Machine (JVM) arguments. The default JVM in Mac OSX (JVM 1.6) was used for running CriticAL. Note that the system memory log showed that only 480 MB of the main memory was used at maximum during the analysis of the two projects.

[2] `http://docs.oracle.com/javase/tutorial/uiswing/examples/components/index.html`
[3] `http://eclipse.org/downloads/`
[4] `http://sf.net/projects/critical/files/updatesite/`

Table 5.2: Performance evaluation of CriticAL on code from the forum and tutorials.

Description	**Statements**	**Time (ms)**	**Loops**	**Paths**
Swing Forum (SF): 90 programs				
Minimum	6	5	0	1
Maximum	265	706	6	2
Average	84.78	108.70	0.34	1.01
Median	68.50	56	0	1
Std. Dev.	54.51	128.11	0.84	0.11
Total	**7630**	**9783**	**31**	**91**
Swing Tutorials (ST): 75 programs				
Minimum	11	3	0	1
Maximum	780	8039	2	13
Average	151.52	184.72	0.20	1.52
Median	110	53	0	1
Std. Dev.	130.97	926.69	0.49	1.54
Total	**11364**	**13854**	**15**	**114**

Table 5.2 summarizes the analysis of the performance for the programs in the two projects. The table shows that even though there is a significant variation in the number of Jimple statements executed per program and the time taken for processing, the number of paths produced and loops unrolled remained quite consistent at *1* (std. dev. 0.11 for SF and 1.54 for ST) and *0* (std. dev 0.84 and 0.49), respectively. Hence, the explosion of paths and processing of loops are not a major issue in processing GUI code for CriticAL. We observed such a behavior because a GUI code is mostly linear in nature that involves composing multiple API elements together through function calls rather than algorithmic code which involves lots of *if* and *switch-case* statements. In total, CriticAL took about 10 seconds to process about 8,000 Jimple statements in SF and about 14 seconds to process about 11,000 statements in ST. Figure 5.2 shows traces of time versus statements during the execution of programs in the two projects. Figure 5.2(a) shows that the majority of programs in SF contained less than 300 Jimple statements. Also, the execution time for the majority of programs was well below 1 second (see Figure 5.2(b)). The program in ST were relatively longer than SF but the majority of them were less than 300 Jimple statements long (Figure 5.2(c)). The execution time for programs in ST were also relatively longer

than SF, however, below 2.5 seconds tops (Figure 5.2(d)).

Even though the total running time for SF and ST were about 10 and 14 seconds, respectively, the actual time from the start to finish for the two projects were approximately 37 seconds and 47 seconds, respectively. The extra time was taken for loading all of the programs in SOOT and for translating the Java byte code to the Jimple IR. 79 % and 76 % of the total time was spent in creating Jimple IR for the programs from SF and ST, respectively, which was the major bottleneck for the performance of CriticAL. Nevertheless, both of the projects were analyzed within a minute. We conclude that CriticAL performed reasonably well in analyzing approximately ten thousand Jimple statements for both benchmark projects (Table 5.2).

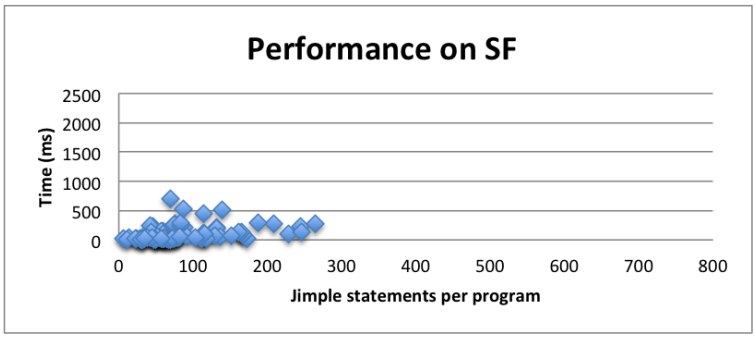
5.2.2 Evaluation of Utility

In Chapter 2, we presented rules that were identified during the manual analysis of the Java Swing forum. In this section, we will summarize the current status of CriticAL in terms of the implementation of those rules and the number of criticisms, recommendations, and explanations generated by CriticAL for the 90 programs collected from the Swing Forum.

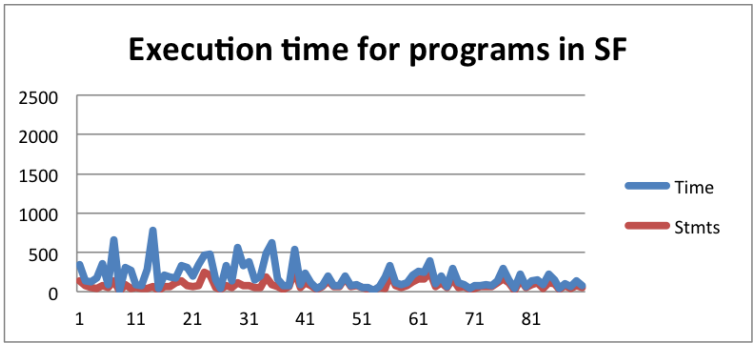
Table 5.3 summarizes the criticisms generated by CriticAL for SF. The *True Pos.* label means true positives and the *False Pos.* mean false positives. The rate of false positives is resonably low at 8.21 %. The code snippet for each rules can be found in Section 2.3 except for he ones marked with a *. We will discuss the three rules added on top of Table 2.1 next and discuss the reason for false positives in Section 5.2.3.

JFrame Invisible: This rule enforces that *after a* `JFrame` *is created, it must be visible before the end of the program*, which can be achieved by calling `JFrame.setVisible(true)`. We have dicussed this rule with an example in Section 1.1.1.

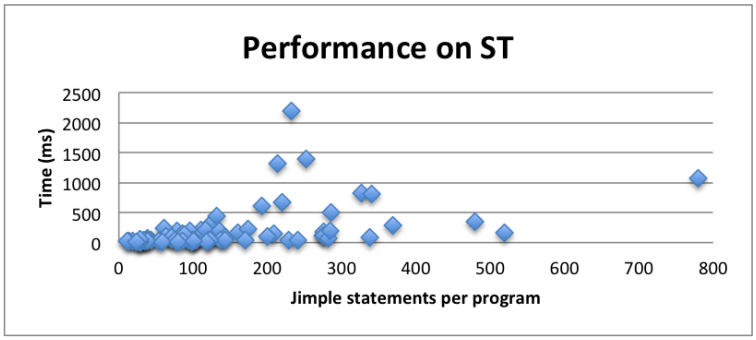
Empty Container: This rule criticizes *having an empty container (*`JPanel`*) whose layout has been explicitly set.* The reason to have a layout manager in a container is to arrange its child widgets properly, hence, when a layout is explicitly set on a container, we require that the user add a widget to the container or not set the layout at all.



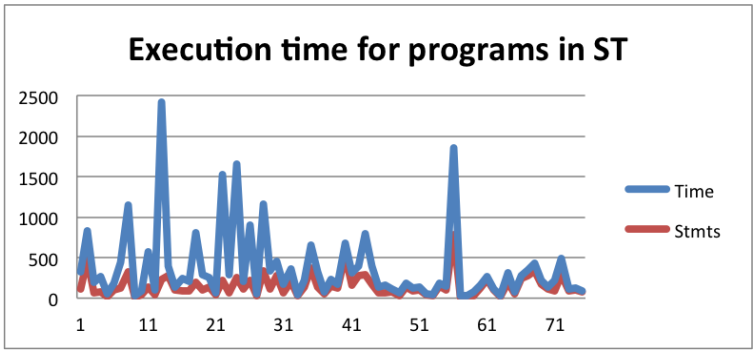
(a) SF - Time vs Statements.



(b) SF - Execution trace of 90 programs.



(c) ST - Time vs Statements.



(d) ST - Execution trace of 75 program.

Figure 5.2: Execution trace of CriticAL on code from the forum and tutorials.

Table 5.3: Evaluation of Criticisms. (Note that * represents the rule that has been added on top of Table 2.1.)

API Criticism Rules	**Implementation**	**True Pos.**	**False Pos.**
Postconditions			
Orphan GUI Objects	Yes	63 (47.01%)	7 (5.22%)
Missing Layout Constraints	No	-	-
Parent Switching	Yes	8 (5.97%)	0
Misplaced Layout Constraints	Partially	0	0
*JFrame Invisible	Yes	1 (0.74%)	4 (2.98%)
*Empty Container	Yes	2 (1.49%)	0
Invariants			
Content Mismatch	Yes	4 (2.98%)	0
Dynamic GUIs	Yes	2 (1.49%)	0
One Layout, One Container	Yes	3 (2.23%)	0
Preconditions			
JFrame.pack() Constraints	Yes	13 (9.70%)	0
Positioning and Sizing Constraints	Yes.	1 (0.74%)	0
*Redundant Action	Yes.	21 (15.67%)	0
Deviation from Usage Conventions			
Components Resizing Behavior	No	-	-
Table Design	Yes	5 (3.73 %)	0
Total		123 (91.79%)	11 (8.20%)

Redundant Action: By default, the content pane of `JFrame` has a `BorderLayout` and a newly created `JPanel` has a `FlowLayout` as its layout manager. We have seen cases where users reset the same kind of layout manager on these widgets, which is redundant and shows a sign of confusion on the part of users. Hence, we criticize this behavior.

Table 5.4 and 5.5 sumarizes the explanations and recommendations generated for the programs in SF. It is natural to see so many explanations and recommendation produced because they do not represent problems in the code but explain the behavior of the code and provide suggestions on the next useful API elements. Hence, to properly assess the quality of explanations and recommendations, we need to perform a summative user case study as a future work. The discussion on rules for explanations and code snippets can be found in Section 2.4 (also see Table 2.2) and for recommendations in Section 2.5 (also see Table 2.3).

Table 5.4: Summary of Explanations.

API Explanation Rules	**Implemented**	**True Pos.**	**False Pos.**
Behavior of Null Layout	Yes	18 (9.42%)	0
Behavior of `GridbagLayout`	No	-	-
Resizing Behavior of `BorderLayout`	No	-	-
API Specific Explanations	Partly	173 (90.57%)	0
Total		**191**	**0**

Table 5.5: Summary of Recommendations.

API Recommendation Rules	**Implemented**	**True Pos.**	**False Pos.**
Generic Recommendations	No	-	-
Syntax-Based Recommendations (Confusing APIs / Lite Context)	Yes	15 (6.25%)	0
State-Based Recommendations (Unused Features / Alternative Design)	Partly	222 (92.50%)	3 (1.25%)
Total		**237 (98.75%)**	**3 (1.25%)**

5.2.3 Reason for False Positives

False positives in CriticAL are produced mostly due to the compromises made in its design. In this section, we will categorize the root causes of false positive and discuss their fixes. Table 5.6 summarizes root causes for false positives. The major cause of false positives for the programs in SF was unsupported API elements. We will discuss each in details.

Table 5.6: Analysis of False Positives. (C: Criticism and R: Recommendation)

Root Causes	**Critiques Type**	**Total**
Entry Point	C (3)	3 (21.42%)
Loop Unrolling	C (2)	2 (14.28%)
Unsupported API Elements	C (5), R(3)	8 (57.14%)
Implementation Bugs	C (1)	1 (7.14%)
Total		**14**

Listing 5.2: An entry point problem.

```
1 public static class TEST {
2  private JFrame frame;
3  public static void main(String[] args) { new TEST(); }
4  public TEST() {
5   makeFrame();
6   frame.pack();
7   frame.setVisible(true);
8  }
9  public void makeFrame() {
10  frame = new JFrame ("Test");
11  Container content = frame.getContentPane();
12  ... // Populate the content pane with other widgets
13 }
14 }
```

Entry Point

CriticAL is a symbolic execution framework. It requires an entry point to start the analysis. In our design of the Swing exetension plugin, we chose the method that initializes a `JFrame` as an entry point for analysis. There was a problem with this approach. Consider the example shown in Listing 5.2 [5]. Even though the `JFrame` is initialized in `makeFrame()`, it is not set to visible until the method returns to the construtor `TEST()` in line 7. However, our execution starts and end within the `makeFrame()` method. CriticAL, thus, complains that the `JFrame` initialized at line 10 is not visible at the end of the program (line 13).

This problem can be solved by using the call graph of the entire program to recursively look for the top-level methods which refers to a `JFrame` in its body. For this example, the method would be `Test()` (line 4). Hence, choosing such a method as an entry point helps solve this issue. The other two false positives in this category can be found in [6] [7]. These two false positives are due to thread invocation related control flow that CriticAL currently does not handle.

Listing 5.3: A loop unrolling problem.

```
1 public static class Fubar3 {
2  private JPanel mainPanel = new JPanel();
3  private JTextField nameField = new JTextField(12);
4  private JTextField ageField = new JTextField(12);
5  private JTextField salaryField = new JTextField(12);
6  private JTextField positionField = new JTextField(12);
7  private JTextField[] fields = {nameField, ageField, salaryField, positionField};
8  public Fubar3() {
9   JPanel componentPanel = new JPanel(new GridLayout(0, 1, 10, 10));
10  for (int i = 0; i < fields.length; i++) {
11   componentPanel.add(fields[i]);
12  }
13  ...
14  mainPanel.add(componentPanel);
15 }
16 private static void createAndShowUI() {
17  JFrame frame = new JFrame("Fubar3");
18  frame.getContentPane().add(new Fubar3().getMainPanel());
19  ...
20  frame.setVisible(true);
21 } ...
22 }
```

Loop Unrolling

By default, CriticAL unrolls a loop only twice. While this approach is enough to expose most of the problems related to GUI widgets, however, it may also introduce false positives. Consider the code snippet in Listing 5.3 [8]. When a new `Fubar3` object is created in line 18, it also creates the four text fields in lines 3-6 and assign them to the `fields` array (line 7). The control then reaches the code in line 10-12, which add all of the text fields to `componentPanel`. The `componentPanel` panel is then added to the `mainPanel`, which is returned to line 18 as a return value of the `getMainPanel()` method. When symbolically executing this piece of code, the loop only gets unrolled twice, thus, adding only `nameField` and `ageField` to `componentPanel`. As a result, when the symbolic execution reaches the end (line 21), CriticAL finds that `salaryField` and `positionField` are orphans

[5] `https://forums.oracle.com/forums/thread.jspa?messageID=5721812`
[6] `https://forums.oracle.com/forums/thread.jspa?messageID=5811399`
[7] `http://forums.oracle.com/forums/thread.jspa?messageID=5881971`
[8] `http://forums.oracle.com/forums/thread.jspa?messageID=5758895`

and criticizes the code, which in fact are false positives.

Currently, CriticAL unrolls a loop (with an open or a known loop condition) only twice. Bounding loop to a defined limit is a well-known strategy in the symbolic execution literature [10, 12]. The false positive for this example could be avoided by not enforcing the unrolling limit for a loop with a known loop condition. Because the length of the array is known, the loop in this example will stop after four iterations. However, when a code has an infinite loop, CriticAL will not return back if this approach is used as a general solution. Hence, as a general solution to this problem, CriticAL could be configured with a number greater than four for the loop unrolling limit. A loop unrolling limit must be chosen carefully to balance accuracy and performance of CriticAL. A lower limit will make analysis faster by producing fewer paths but at the cost of precision. Choosing a higher limit will make the analysis more precise but CriticAL will run longer and at times may even suffer path explosion. Nevertheless, we only found two false-positives due to loop unrolling in the same source file and hence, the current limit is good enough for the majority of user's code in SF.

Unsupported API Elements and Implementation Bugs

While we tried to support most of the important API methods related to layout and GUI components, we did not cover them all. We have a few cases where the reason for false-positives are the unsupported API methods. Consider the code snippet in Listing 5.4 [9]. The `mainFrame.add()` method in line 8 actually adds `fatPanel` to the content pane of the `JFrame`. Since we did not support the `JFrame.add()` method, CriticAL found `mainFrame` to be empty at the end of the program and produced a recommendation on adding new widgets to the frame. Furthermore, one recommendation is also produced on how to add widgets to an empty `JFrame` due to this problem. Similarly, we do not have a good support for the API methods of `JWindow`, which also contributed to other four false-positives in this category for criticisms and two for recommendations [10].

[9] `http://forums.oracle.com/forums/thread.jspa?messageID=5694617`
[10] `http://forums.oracle.com/forums/thread.jspa?messageID=9281019`

Listing 5.4: A problem due to an unsupported API method.

```
1 public static class GuiExample extends JFrame {
2  private JFrame mainFrame = new JFrame();
3  private FatherPanel fatPanel;
4
5  public void makeUI() {
6   fatPanel = new FatherPanel();
7   ...
8   mainFrame.add(fatPanel, BorderLayout.CENTER);
9   mainFrame.pack();
10  mainFrame.setVisible(true);
11 }
12 }
```

These issues can be resolved by implementing a proper support for these API elements. After testing CriticAL on multiple projects, we hope to collect more of such corner cases and implement support for them in the future. Finally, the last false-positive in Table 5.6 is produced due to an implementation bug, which can be fixed through debugging and will not be discussed further.

5.3 Efforts in Implementing API Rules

Typically, implementing a rule and creating a critique takes from 10-15 lines of code. The behavior as well as all of the rules involved with `JComponent` have been implemented in less than 1000 lines of code and that of `JFrame` have been implemented in less than 500 lines of code. All of the other components have been implemented in less than 100 lines of code each. Hence, we can safely conclude that supporting an API through the CriticAL framework is a relatively simple task. However, finding a rule that needs to be enforced may be a time consuming process as we experienced in Chapter 2.

5.4 Conclusion

We have evaluated CriticAL using a formative user study as well as through users' programs collected from the Java Swing forum. It has produced many useful critiques for the code

written by novices within a matter of a few seconds and with a low rate of false positives. Nevertheless, the presence of false positives may confuse a novice programmer. Hence, to properly help a programmer, the advice generated by CriticAL should not only mention about the problems but also provide an overview of the rule and the context in which the advice was generated. This approach may help a programmer decide whether the generated critiques are real problems or false positives.

Chapter 6

Related Work

6.1 Study of the API-Usability Problem

Ko et al have identified six learning barriers in their study of 40 novice programmers learning Visual Basic [43]. These barriers take account of several factors such as design, selection, coordination, use, understanding, and the availability of information about APIs. Based on their observations, they make recommendations for improving end-user programming systems by providing more examples, making search experience better, making the invisible system's rules more visible through errors messages from tools, making the development environment more interactive, and developing tools that could explain some of the complex behavior of the APIs to the novices. The recommendations made by Ko et al align closely with the goals for the Critic system proposed by Fischer et al in [17] and to that of ours for CriticAL [53].

A case study with the Swing Forum in [33] reinforces Ko's learning barriers for the programmers of the Java Swing API. Furthermore, Robillard's qualitative analysis of the API learning difficulties percieved by Microsoft's developers [48, 49]; Hou's quantitative analysis of framework learning difficulties for undergraduate students [31]; Ko's qualitative studies on the role of conceptual knowledge about frameworks [41]; and other case studies of the Swing framework [34,35]; all point to the importance of design knowledge about frameworks

for the proper reuse of APIs. Some of the challenges reported in Robillard el al studies arise from the lack of resources such as online API documentation and code snippets, inadequate design documentation, complicated API design, and the lack of background knowledge of the systems. They emphasize on documenting the intent behind APIs, providing code examples containing several matching API elements and usage scenarios, and properly formating the API documentation for clear presentation. In contrast, we attempt to connect the design knowledge about a framework to programmers at a fine grained level of framework rules identified through our case study research in Chapter 2.

6.2 API Critic

A reuse-based system may not be used in the best possible ways that it is originally designed for. Instead, users often settle for a suboptimal set of available solutions that are just enough for their current tasks [17]. There are several causes to this problem. A user may not know what functionalities are available, what is the best solution among multiple alternatives, how he can use these functionalities, or how he can combine, adapt, and modify them. To address this problem, Fischer envisions an architectural design for a development environment that encompasses such tools as visualization, explanation, recommendation, and critics, to help programmers work with the framework and APIs [17]. Several critics have been developed including one for the Lisp programming language [19–22]. CriticAL, is the first for the Java programming language.

Like CriticAL, the Lisp critic facilitate incremental learning and support learning on demand [18]. It also requires rules and documentation to produce useful advice. However, unlike CriticAL, the rules are matched using structure of the program (or syntax). CriticAL models the state or behavior of a Java program, which makes it more resilient to syntactic variations, thus, requiring less number of rules to attack problems that share the same semantics but different syntactic patterns.

6.3 Related API Tools

Extensive past work has been done in searching [7, 26, 60], explaining and exploring examples [19, 47, 51], and understanding and debugging [42]. While understanding framework design is important, it is also important to find the right API and API elements for the programming task. Code search tools such as Blueprint [3] brings code snippets to the IDE based on search keywords. Other tools [26, 29, 60] help programmers locate documentation, examples, and related projects on the web. Nevertheless, novice programmers often find it hard to formulate a useful search query to get the relevant results [41] and to assess the quality of the results. We, on the other hand, do not require any keywords from programmers and directly work on the available source code to produce relevant suggestions for their programming needs.

The main functionality of the search tools is to find code snippets and documents on the basis of key words. Our tool should complement them. By adding the elements of code pattern recognition and program understanding, it could better predict the relevancy of a document to the code under analysis. But unlike theirs, we require the library providers to write rules for API uses, which may be a burden. However, we have seen in Chapter 5 that having a set of few potent rules can impact a large number of users' code. Hence, such an investment can be justified in the long run.

Tools such as MAPO [63] perform sequence analysis of API elements and could be used to recommend the next set of API elements to novice programmers. Nevertheless, such a tool does not make design inferences and may suggest APIs that are not relevant to the problem, thus, further confusing them. Furthermore, tools for explaining program behavior through multiple views have also been researched previously [47]. However, none of these tools provide a unified way to present explanations, criticisms, and recommendations for APIs that CriticAL provides.

6.4 Symbolic Execution

Use of symbolic objects for program testing was first proposed by King [40]. He envisioned using symbols instead of real inputs in a program and executing it to test important properties of the program. The symbolic inputs would represent a class of inputs rather than sample inputs and would be equivalent to a large number of normal test cases. He classified symbolic execution as a compromise between program verification where all inputs are considered and testing where fixed inputs are considered.

Cousot et al. unify static analysis techniques including symbolic execution under abstract interpretation in [6]. They describe the behavior of a program as a computation of objects within a certain universe. They then describe an abstract interpretation of the program as a computation of abstract objects in another universe whose outcome gives certain information about the actual computation. The results of abstract interpretation over-approximate the concrete program behavior and thus, are inherently incomplete but useful.

Symbolic execution has been used by many for bounded program verification and model checking such as in Forge [12], Kiassan [10], and Java Path Finder (JPF) [38]. Typically such tools accept declarative specification in the form of program annotations. All of the path conditions are expressed as constraints on symbolic variables and are conjoined with specification and passed to a model checker. Since their job is to verify strong properties of a program, they become infeasible as the client code increases in size and uses large libraries. Khursid et al. proposed an approach for abstracting the behavior of library code to make the symbolic execution more scalable in verification of the client code in the Dianju tool [39]. The supported library classes were *String, Set,* and *Map.* The tool, however, suffered problems with array manipulation because the symbolic execution of an array index represented by an integer would refer to several array elements introducing a path explosion during path exploration. Due to the linear nature of GUI-based code we did not encounter these problems in our test cases. Nevertheless, CriticAL has not been tested in a large code base to concretely reason about such problems.

Grechanik proposes a symbolic execution tool called *Viola* in his PhD dissertation to find problems with interoperating components [25]. The tool checks whether the XML parsing code used in two interoperating languages (C++ and Java) are based on a common XML schema. This comparison is done after a series of pipelined processes are executed. After extracting execution paths from a source code, the tool abstracts away the code not related to XML processing. All of the API elements related to XML processing are then translated to abstract operations: *Navigate, Read, Write, Add, Delete, Load*, and *Save*. *Viola* symbolically executes these abstract operations to generate the corresponding schema. A schema comparator algorithm is used to check if the generated schemas for the code written in the two languages match with each other and to the actual schema. Any discrepancies are reported to the user. *Viola* is a domain specific tool tailored for the XML processing code and hence, may not be generalizable. CriticAL models the return value of an unsupported API method using an open (non-deterministic) symbolic object. This approach lets CriticAL do away with expanding such an API call. Different from *Viola*, it generalizes the abstract operations to support more than just XML API methods or just Swing methods. The abstraction for an API method is user defined and can be specified through an extension plugin. Nevertheless, *Viola* and CriticAL share a similar technique for bounding loops and recursive functions. The problem with the entry point selection, however, is not clearly discussed in the dissertation.

GUI-based code uses call back methods or listeners to provide event-based functionalities, e.g., action listeners for buttons, which are not part of the main control-flow. We inline the call back methods registered to the GUI widgets at the end of the path. The ordering of user events may not be known statically hence we derive a combination of up to k (configurable) callback methods at a time for inlining. i.e. if n listeners have been registered in a path, we produce $\binom{n}{k}$ different listener sequences and inline them at the end of the path. Note that the current implementation, however, only supports inlining one listener method at a time. This technique has also been used by others in the generation of GUI test-cases [24, 45]. A survey of symbolic execution techniques can be found in [46]

Different from all of these verification tools that use symbolic execution, our prime goal is advice generation, rather than program verification. In fact, CriticAL is the first use of the symbolic execution technique to actively help programmers in the programming process rather than just finding bugs in their code.

6.5 Static Analysis

We have borrowed some of the techniques described in [1, 57, 58] to achieve path-sensitive, inter-procedural program analyses. Our analysis is not only path sensitive but also state-sensitive, i.e. each object knows its state at every control point. We have used path-sensitive static analysis previously in a model-finding tool called *EQ* [1] [52, 56]. The tool can identify the low-level programming errors such as *NullPointerException* and *ClassCastException* using its path-sensitive, inter-procedural program analysis framework built on top of the *Jimple* intermediate representation [61]. It further uses these paths to formulate a high-level logical model in Alloy [37]. The Alloy model checker is then invoked automatically to detect the semantic problems with the equivalence property of the Java classes. Other program analysis tools such as FindBugs [36], SCL [32], Design Fragments [16], and Athena [44] only provide criticisms for the design and implementation bugs and do not focus on recommendations and explanations.

FindBugs [36] is a flow-sensitive, intra-procedural program analysis tool. It uses several data-flow analyses to accumulate the information about a code. Error patterns can be programmatically specified using the result of the flow analyses as an extension to the tool. CriticAL, on the other hand, is a path-sensitive, inter-procedural program analysis framework that relies on symbolic execution.

SCL [32] (and its precursor FCL [30]) share the same vision as that of CriticAL, i.e. to support API-based programming. SCL is a structure-based program analysis tool that checks for a contract violation based on the syntax of the API-client code. The same is true for Design Fragments [16]. CriticAL, on the other hand, is a behavior based tool and

[1] http://eqchecker.sourceforge.net/

models the program state abstractly to reason about the API-client code.

Athena [44], a path-sensitive, inter-procedural program analysis tool developed by Le and Sofa helps detect programming faults along a path. They make the observation that not all of the programming paths are relevant to the faults that need to be detected and not all of the statements in the path contribute to the faults of interest. Hence, they apply a demand-driven strategy to fault detection based on dependencies between program variables that helps them explore only the paths that are relevant to the faults of interest. CriticAL models the return value of an irrelevant API method as an open-symbolic object, thus, doing away with expanding the method. Nevertheless, it symbolically executes all of the paths from an entry method. Entry methods can be configured in CriticAL to slice away program segments that are irrelevant to the properties of interest. This strategy helps CriticAL focus only on the code that needs to be supported.

6.6 Dynamic Analysis

CriticAL is closely related to the Whyline tool [42] in that both tools reason about program states. However, while Whyline answers pre-defined *"why did"* and *"why didn't"* questions by tracking the concrete values of program variables and their causalities through dynamic execution, CriticAL offers greater flexibility in terms of defining the possible API usage rules, hence has stronger capabilities in criticizing and explaining API behavior, and in producing recommendations. Furthermore, Whyline reasons about one execution trace at a time and, thus, requires multiple runs to have enough branch coverage, whereas CritcAL symbolically executes all possible execution traces in a single run.

6.7 Program Verification

Program verification techniques encode either programs [13, 62] or path conditions as logic formulae [11, 12, 38] and solve the verification problem with a constraint solver or a theorem prover. Other verification techniques include theorem-prover-based approaches such as

ESC/Java [23]. These techniques specify the pre and post conditions for a function and check if the post condition holds at the end of the function. Other techniques use notation such as LTL to reason about the temporal properties of a program using a model checker [2, 4, 5, 27].

CriticAL can also be used for checking the properties of a program in conceptually the same way as pre/post condition based checking is done in program verification tools. The pre and post conditions for CriticAL would be predicates whose arguments would be symbolic objects and specified using the Java programming language itself. However, a constraint solving module needs to be implemented in CriticAL for curbing false positives if used as a verification tool. This application could be one of the future directions for CriticAL, however, the current focus is not only in finding problems but to also have an engine [50] that could provide explanations and recommendations for API-client code. Hence, CriticAL differs from program verification tools in the problem domain.

Das et al. propose a path-sensitive program verification tool called ESP to check the temporal properties of a program [8]. They only model the branches that are relevant to the properties being checked in a program. Using this approach, they check the temporal properties of the file IO API in GCC. CriticAL could also achieve a similar effect by abstractly specifying the state of the IO streams through the corresponding open/close API methods. CriticAL, however, does not merge the state of a property group as done in ESP, thus, avoiding unsoundness resulting from the merging operation. Nevertheless, merging a property group helps curb the path-explosion problem and reduces the memory footprint. Because of this feature, ESP enumerates paths in the breadth-first fashion using a worklist algorithm [8]. CriticAL, however, enumerates paths in the depth-first fashion and releases the resources after a path has been completely processed. On encountering an unknown branch condition, CriticAL clones the stack, however, sharing the symbolic objects between the clones. Each shared symbolic object is cloned lazily just before a write operation. This strategy helps CriticAL reduce the memory footprint during the analysis.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We saw that learning to use API is an intense process and comes with several challenges. To cope with their difficulties in *finding*, *understanding*, and *debugging* API-based code, we presented CriticAL that offered them with *recommendations*, *explanations*, and *criticisms*, respectively. CriticAL took API usage rules as input, performed symbolic execution to check that the client code had followed those rules properly, and generated advice as output to help improve the client code.

Several framework rules were collected as a result of the case study of the Java Swing forum. The study also showed that the API usage problems recur in practice and tool support can be justified. Most of the rules collected from the case study were implemented in CriticAL and tested with 90 users' programs from the Swing forum and 75 programs from the Swing tutorials. The evaluation showed encouraging result about the usefulness of CriticAL with a low rate of false positives at 8.21%.

We foresee the use of CriticAL in both academia and industry. CriticAL could be actively used in teaching new frameworks and libraries to students. It would complement an instructor by serving as a programming assistant for the novice students when the instructor is not available. Software industries who develop complicated frameworks and

libraries may also ship CriticAL as a part of their library bundle to help new programmers learn their frameworks faster. Instead of the old-fashioned help documentation, CriticAL would provide a contextual help to the programmers during the development phase.

Future research on CriticAL should be directed towards assessing the quality of critiques by conducting a summative case study and towards generalizing CriticAL by supporting more APIs and libraries.

7.2 Future Work

We saw some encouraging results of using CriticAL in Chapter 5. However, we feel that there is some room for improvement. We will discuss the improvement for CriticAL in the next few subsections.

7.2.1 Extending Swing Support

CriticAL can be further strengthened by adding more rules and supporting more API elements in Java Swing. One interesting direction is supporting the resizing behavior of components in the presence of a layout manager. Such a support would require us to statically reason about the size and location of each GUI component laid out by the layout manager. Since there could be several arrangements of components as governed by layout constraints, modeling those constraints using a constraint solver may help reason about the layout and resizing behavior of the component more precisely. Research in this direction may not only help in producing valuable criticisms but also in recommending alternative solutions, such as recommending a more appropriate layout manager for the user's task.

7.2.2 Conducting A Summative User-Case Study

We have evaluated CriticAL for utility using the observational user case study method [28] and from the source code collected from the Java Swing forum. Even though we now have some idea about the usefulness of CriticAL based on the formative case study and our regular experience with CriticAL, we have not yet thoroughly evaluated CriticAL for

usability. We should further assess CriticAL in terms of learnability (should be easy to learn), memorability (should be easy to remember previously done actions), errors (should not crash or should have fewer errors), and satisfaction (user should find it pleasant to use CriticAL) in the future. We need to perform a summative evaluation through a set of questionnaires about users' experience with CriticAL so that we can properly understand users thoughts and experience about the quality of critiques produced by the tool.

7.2.3 Testing Generalizability of CriticAL

Another future direction for CriticAL would be to test its generalizability by extending support to other APIs such as SWT, JDBC, JMX, and the Android platform. The SWT toolkit is used for designing GUIs in Eclipse and uses similar design concepts as Swing. Hence, an immediate extension of CriticAL could be SWT. In principle, CriticAL operates on top of Jimple IR and should be able to support any language, API, or libraries that can be translated to Jimple IR.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006, pp. 903–964.

[2] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," *SIGPLAN Not.*, vol. 37, pp. 1–3, 2002.

[3] 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.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new Symbolic Model Verifier," in *CAV*, 1999, pp. 495–499.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from java source code," in *ICSE*, 2000, pp. 439–448.

[6] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.

[7] B. Dagenais and H. Ossher, "Automatically locating framework extension examples," in *FSE*, 2008, pp. 203–213.

[8] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive Program Verification in Polynomial Time," in *PLDI*, 2002, pp. 57–68.

[9] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," in *ECOOP '95*, 1995, pp. 77–101.

[10] X. Deng, "Contract-based verification and test case generation for open systems," Ph.D. dissertation, Kansas State University, Manhattan, KS, USA, 2007.

[11] X. Deng, J. Lee, and Robby, "Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *ASE*, 2006, pp. 157–166.

[12] G. D. Dennis, "A relational framework for bounded program verification," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009.

[13] J. Dolby, M. Vaziri, and F. Tip, "Finding bugs efficiently with a SAT solver," in *ESEC-FSE*, 2007, pp. 195–204.

[14] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for dpll(t)," in *CAV*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 81–94.

[15] K. M. Eisenhardt, "Building Theories from Case Study Research," *Academy of Management Review*, vol. 14, no. 4, pp. 532–550, 1989.

[16] G. Fairbanks, D. Garlan, and W. Scherlis, "Design Fragments Make Using Frameworks Easier," in *OOPSLA*, 2006, pp. 75–88.

[17] G. Fischer, "Cognitive view of reuse and redesign," *IEEE Softw.*, vol. 4, pp. 60–72, July 1987.

[18] ——, "A critic for lisp," in *Proceedings of the 10th international joint conference on Artificial intelligence - Volume 1*, ser. IJCAI, 1987, pp. 177–184.

[19] G. Fischer, S. Henninger, and D. Redmiles, "Cognitive tools for locating and comprehending software objects for reuse," in *ICSE*, 1991, pp. 318–328.

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

[21] ——, "The role of critiquing in cooperative problem solving," *ACM Trans. Inf. Syst.*, vol. 9, no. 2, pp. 123–151, 1991.

[22] G. Fischer, K. Nakakoji, J. Ostwald, G. Stahl, and T. Sumner, "Embedding computer-based critics in the contexts of design," in *CHI*, 1993, pp. 157–164.

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *PLDI*, 2002, pp. 234–245.

[24] 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.

[25] M. Grechanik, "Design and analysis of interoperating components," Ph.D. dissertation, University of Texas at Austin, Austin, TX, USA, 2007.

[26] 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.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," *SIGPLAN Not.*, vol. 37, pp. 58–70, 2002.

[28] D. M. Hilbert and D. F. Redmiles, "Extracting usability information from user interface events," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 384–421, Dec. 2000.

[29] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: finding and leveraging implicit references in a web search interface for programmers," in *UIST*, 2007, pp. 13–22.

[30] D. Hou, "Fcl: automatically detecting structural errors in framework-based development," Ph.D. dissertation, University of Alberta, Edmonton, Alta., Canada, 2004.

[31] ——, "Investigating the effects of framework design knowledge in example-based framework learning," in *ICSM*, 2008, pp. 37–46.

[32] D. Hou and H. J. Hoover, "Using SCL to Specify and Check Design Intent in Source Code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 404–423, 2006.

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

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

[35] D. Hou, K. Wong, and H. J. Hoover, "What can programmer questions tell us about frameworks?" in *IWPC*, 2005, pp. 87–96.

[36] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion of OOPSLA 2004*, 2004, onward! track.

[37] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[38] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS*, 2003, pp. 553–568.

[39] S. Khurshid and Y. L. Suen, "Generalizing symbolic execution to library classes," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 103–110, September 2005.

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

[41] A. J. Ko and Y. Riche, "The role of conceptual knowledge in API usability," in *VL/HCC*, 2011, pp. 173–176.

[42] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *ICSE*, 2008, pp. 301–310.

[43] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *VL/HCC*, 2004, pp. 199–206.

[44] W. Le and M. L. Soffa, "Generating analyses for detecting faults in path segments," in *ISSTA*, 2011, pp. 320–330.

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

[46] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 4, pp. 339–353, Oct. 2009.

[47] D. F. Redmiles, "Reducing the variability of programmers' performance through explained examples," in *CHI*, 1993, pp. 67–73.

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

[49] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Softw. Engg.*, vol. 16, pp. 703–732, 2011.

[50] M. P. Robillard, R. J. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, July/August 2010.

[51] M. B. Rosson, J. M. Carroll, and C. Sweeney, "A view matcher for reusing smalltalk classes," in *CHI*, 1991, pp. 277–283.

[52] C. R. Rupakheti and D. Hou, "An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java," in *WCRE*, 2010, pp. 205–214.

[53] ——, "Satisfying programmers' information needs in API-based programming," in *ICPC*, 2011, pp. 250–253.

[54] ——, "CriticAL: A Critic for APIs and Libraries," in *ICPC*, 2012, 10 pp. (to appear).

[55] ——, "Evaluating Forum Discussions to Inform the Design of an API Critic," in *ICPC*, 2012, 10 pp. (to appear).

[56] ——, "EQ: Checking the implementation of equality in Java," in *ICSM*, 2011, pp. 590–593.

[57] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *CC'03*, 2003, pp. 126–137.

[58] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1981, pp. 189–233.

[59] B. G. Silverman, "Survey of expert critiquing systems: Practical and theoretical frontiers," *Commun. ACM*, vol. 35, no. 4, pp. 106–127, 1992.

[60] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *VLHCC*, 2006, pp. 195–202.

[61] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *CASCON*, 1999, pp. 125–135.

[62] M. Vaziri-Farahani, "Finding bugs in software with a constraint solver," Ph.D. dissertation, MIT, MA, USA, 2004.

[63] T. Xie and J. Pei, "MAPO: mining API usages from open source repositories," in *MSR*, 2006, pp. 54–57.