AMERICAN UNIVERSITY OF BEIRUT

GUICop: Specification Based GUI Testing

by

Dalal Samir Hammoud

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering
to the Department of Electrical and Computer Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
September 2015

AMERICAN UNIVERSITY OF BEIRUT

GUICop: Specification Based GUI Testing

by

Dalal Samir Hammoud

Approved by:

Dr. Fadi Zaraket, Assistant Professor	Advisor
Electrical and Computer Engineering	[illegible]
Dr. Ali Chehab, Professor	Member of Committee
Electrical and Computer Engineering	Ali Chehab
Dr. Wassim Masri, Associate Professor	Member of Committee
Electrical and Computer Engineering	[illegible]

Date of thesis defense: September 11, 2015

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name:	Hammoud	Dalal	Samir
	Last	First	Middle

[✓ Master's Thesis] ◯ Master's Project ◯ Doctoral Dissertation

☒ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project,** to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

[illegible]
Signature

September 22, 2015
Date

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Fadi Zaraket for his continuous support, motivation, patience and for his huge knowledge and expertise. He has been a great guidance through it all and his constant advice has immensely and positively shaped the way I think across the years of my Masters. I was very fortunate to have such a distinguished advisor who gave me the freedom of exploring and researching on my own, and at the same time the immediate help and support whenever I was tripping. Fadi taught me how to organize my thoughts and ideas and become able to express them clearly and never failed at giving me the best advice whenever I'm in distress. I never worried about having any problems especially technical ones, because I was confident that he would have the solution. I'm thankful for the countless times he encouraged me to use correct grammar and notations and to be consistent in the write up of my thesis. I'm thankful for him being always understanding and compassionate in a way that I never felt estranged. In short, I couldn't have asked for a better advisor.

Besides my advisor, I would like to thank my co-advisor Dr. Wassim Masri for being patient and motivating and always supporting my work and following up regularly on my progress. Dr. Masri has been always there to help me sort out the technical problems of my work and give me the most insightful comments. Fadi and Wassim are a great team and their contribution to any additional member is tremendous. I am thrilled that I was a part of this team over the last two years.

Dr. Ali Chehab is one of the best teachers I have had throughout my years at AUB. His questions and comments during my comprehensive exam were thought provoking and helped me enrich my ideas and improve many areas in my work and research.

My friends Mario Achkar, Abbas Bazzi and Farah Charab and many others have kept me in check all the time and helped me stay calm and sane throughout these years. Whenever I'm in distress, I find them by my side supporting me and helping me overcome all sorts of problems. I greatly value their friendship.

Last but not the least, this would have never gone through without the care, the appreciation and support of my family. I would like to thank them for believing in me.

An Abstract of the Thesis of

Dalal Samir Hammoud for Master of Engineering
Major: Electrical and Computer Engineering

Title: GUICop: Specification Based GUI Testing

Oracles used for testing *graphical user interface* (GUI) programs are required to take into consideration complex non-functional factors such as variations in screen resolution or color scheme. To accommodate this aspect of GUI testing, specifically when comparing observed to expected outputs, researchers proposed fuzzy comparison rules and computationally expensive image processing techniques to tame the comparison process; which is necessary, otherwise absolute comparison would be too conservative to be practical.

Alternatively, we propose GUICop, a new approach with a supporting toolset that checks whether the execution trace of a GUI program adheres to a user-defined specification that is expectedly free of non-functional aspect. GUICop comprises the following: 1) a *GUI specification language*; 2) *instrumented GUI libraries*; 3) a *solver*; 4) a *driver*; and 5) a *code weaver*. The user defines the functional specifications of the subject application using the *GUI specification language* whose alphabet consists of: a) basic geometric objects describing GUI components; b) GUI events; and c) positional operators that express relative object positions. The *driver* traverses the GUI structure of the subject and generates events that drive its execution. The *GUI libraries* capture the GUI execution trace, i.e., information about the relative positions taken by the displayed GUI components. And the *solver*, enabled by the *code weaver*, checks whether the traces satisfy the specifications. We successfully evaluated GUICop using case scenarios that we developed and real life case studies such as JEdit, Advanced JukeBox and Gason.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Testing of *Graphical user interface* (GUI) applications entails many challenges not presented when testing command line applications. For example, legitimate GUI test inputs might consist of long sequences of various types of GUI events as opposed to a fixed number of parameters with predefined types. Also, when auditing the results of a GUI execution, judging whether the expected behavior is observed is by no means trivial, i.e., precise and/or sensible oracles are hard to construct.

Some major complicating factors for developing GUI oracles are actually non-functional in nature, such as variations in screen resolution, color scheme, and line attributes (e.g., style, thickness, transparency etc.). To alleviate the impact of these factors, researchers proposed fuzzy comparison rules [1] and computationally expensive image processing techniques [2][3] to tame the process of comparing observed GUI outputs to expected outputs. This is necessary; otherwise absolute comparison would be too conservative to be practical. It should be noted that many researchers opted to totally circumvent this issue by relying on the null-oracle, which simply considers a program to have failed if it terminates abnormally or never terminates [4]. Alternatively, we propose GUICop, a new approach and a supporting tool set that checks whether the execution trace of a GUI program adheres to its user-defined specification which is expectedly free of any non-functional aspects. In order to support specification-based GUI testing, we devised GUICop to include the following: 1) a *GUI specification language*; 2) a *driver*; 3) *instrumented GUI libraries*; 4) a *solver*; and 5) a *code weaver*. The user defines the functional specifications of the subject application using the GUI specification language whose alphabet consists of: a) basic geometric objects describing GUI components; b) GUI events; c) positional operators that express relative object positions; and d) temporal operators that express event timings. The *driver* traverses the GUI structure of the subject and generates events that drive its execution [5]. The GUI libraries capture the GUI execution trace [6], i.e., information about the relative positions taken by the displayed GUI components and the times when the GUI events were triggered. And the solver, enabled by the code weaver, checks whether the traces satisfy the specifications. A sizable body of work on GUI testing was conducted in the past two decades. The most notable was the work of Memon et al. [7][8][9][10][11][12], which we describe subsequently. In fact, the GUICop driver leverages part of that work [5]. Abbot[1] is an existing specification-based GUI testing framework that is an extension of JUnit. It supports writing specifications for programmable Java GUI components, but unlike GUICop, stops short of enabling the user to specify general layout and component interactions. For example, a component may match its programmable specification, even if it was partially hidden by another component on the screen. Other JUnit extensions that enable the user to write assertions also suffer from

[1]abbot.sourceforge.net

that problem, namely, JFCUnit[2], Pounder[3], Marathon[4], SWTBot[5], UISpec4J[6], and Jemmy[7]. In most cases, the checks enabled by the aforementioned tools rely on a hierarchical tree structure where the nodes are the GUI components such as frames, text boxes, and push buttons, and the edges represent parenthood relations. Generally, these tools take the following steps: (1) finding a GUI component of interest starting from the root of the GUI tree based on the programmable name of the component; (2) exercising a relevant event on the component; and (3) checking the status of the GUI tree following the event using JUnit assertions. However, unlike GUICop, the above steps suffer from the following problems:

- **Programmable component names are not always known**. Most developers do not always name their GUI components. And even if they do, the names are not necessarily known by testers. Also GUI components could be automatically generated, for example, a scroll bar in an edit box gets instantiated when the length of the text exceeds the width of the edit box.

- **GUI trees are not adequately expressive**. GUI trees capture parenthood information amongst visible components, i.e., they express positional containment only and fall short of expressing other positional and timing relations. For example, a YES/NO dialog box may contain the title bar, the message label box, and the YES/NO push button components. While it is easy to express and check such containment relation using a GUI tree, it is not possible to express and check the layout of the components, e.g. YES

[2]jfcunit.sourceforge.net
[3]pounder.sourceforge.net
[4]marathontesting.com
[5]swtbot.org
[6]www.uispec4j.org
[7]jemmy.java.net

is to the left of NO.

This paper makes the following contributions:

- A new GUI specification-based testing approach and supporting tool set that circumvents non-functional discrepancies that hinder the task of reusing GUI test suites, such as changes in screen resolution.
- A novel specification language that enables capturing layout information of GUI components, and timings of GUI events.
- A solver that monitors a GUI execution via instrumentation and code weaving, in order to check whether the GUI application satisfies its user-defined specification.
- An extendible library of specifications of common GUI components, which allows for the reuse of specifications.
- The proposed GUI specifications could provide the basis for automated generation of sophisticated test cases that exercise complex GUI functionality and layouts.

The remainder of this work is organized as follows. Chapter 2 offers a background about several important notions. Chapter 3 motivates the work. Chapter 4 discusses related work. GUICop and its components are described in Chapters 5, 6, 7, 8, 9 and 10. We present our case studies in Chapter 11. Finally, we conclude in Chapter 12.

Chapter 2

BACKGROUND

2.1 Java Swing

2.1.1 Introduction

Swing is a GUI API, the primary one for the Java platform and part of Oracle's Java Foundation Classes (JFC). It offers a wide set of components to build desktop interfaces. The set of components ranges from simple elements such as windows, buttons, scrollbars, frames etc., to more complex objects such as file choosers, combo boxes and menus [13].

The Swing classes are built on top of the Abstract Window Toolkit (AWT) architecture. AWT is Java's original platform-independent GUI API. It addressed the portability challenges that windowing systems presented earlier. AWT provided the magic of maintaining the look and feel corresponding to a specific environment: one can create a program on one platform and deliver the compilation output (byte-codes/class files) to every other supported environment without recompilation [14].

This powerful property was maintained to a certain extent in Swing which differs from AWT by having a more sophisticated set of GUI components. Swing provides a native look and feel that imitates the look and feel of several platforms although a small difference is still distinguishable. Swing also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. Swing classes are solely written in Java with no native code while AWT wraps native GUI components. Swing draws its own components by using Java 2D to call low level operating system drawing routines. Among the advanced components Swing provides, there are tabbed panels, scroll panes, trees, tables and lists.

2.1.2 Architecture

Swing components are loosely based on the traditional Model-View-Controller (MVC) programming paradigm [15]. A Model encompasses the state data for each component. For instance a model of a scrollbar may have information about the current position of its thumb, how wide the thumb should be, while a view is its visual representation; how you see it on the screen. The view calculates the position and width of the thumb in screen pixels. A controller takes the input from the user on the view and reflects the changes in the data; it dictates how components react to events such as mouse clicks, gaining or losing focus. It knows for instance that dragging the thumb of a scrollbar is a legitimate action and that the action of dragging should stop at the limits of the endpoints [16]. The diagram in Figure 2.1 shows an architectural perspective of the MVC pattern. Swing uses a simplified version of MVC, the model-delegate, where the view and the controller are combined into a single element, the UI delegate [16]. In this way, Swing has a pluggable look-and-feel architecture.

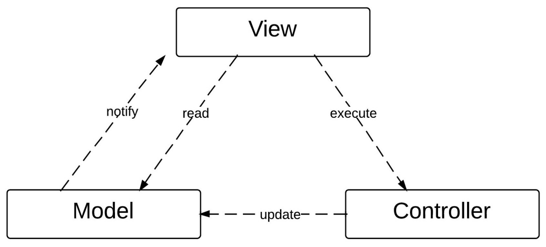


Figure 2.1: MVC pattern.

2.1.3 Coordinate Spaces

Graphics2D objects have their coordinates specified in a coordinate system called User Space. The other coordinate system is device-dependent and varies according to the target rendering device. Transformations from User to Device Space are represented by `AffineTransform` objects responsible for scaling, translating & rotating objects [17].

The User Space origin is located in the upper-left corner of the space, with x values increasing to the right and y values increasing downward [18]. User coordinates are transformed to a virtual device space that approximates the resolution of the target device [17].

2.2 ANTLR

ANTLR is a sophisticated parser generator used to implement language interpreters, compilers, tree walkers and other translators [19]. ANTLR is used to build interpreters for domain specific languages. Once ANTLR is provided with the language grammar, it automatically generates the lexical analyzer (lexer) and parser by analyzing the provided grammar. The lexical analysis operates on the incoming character stream. The parsing phase operates on a stream of vocabulary symbols, called tokens, produced by the lexical analyzer.

A typical ANTLR grammar is a combined grammar that specifies both the parser and lexer rules. Any set of strings that can be defined using tokens (concatenated characters, alternation of characters, or repetition of character an arbitrary number of times) is called a regular set and is recognized by scanners. Any set of strings that can be defined in terms of tokens or recursion is called a context-free language. It is recognized by parsers.

An Abstract Syntax Tree (AST) can be generated by the parser. It is an essential structure in GUICop. An AST is contrasted with a parse tree which represents the syntactical form of the input. The AST records meaningful tokens and the grammatical structure used by the parser. It omits rule names, punctuation and white spaces.

2.3 Aspect Oriented Programming

Object Oriented Programming (OOP), one of the most important programming paradigms, faces an important limitation in software development. It is essentially static, and any change in requirements can have a huge effect on development timelines [20].

Using Aspect Oriented Programming (AOP), the developer can dynamically modify the OO model to meet new requirements without affecting the original code. The AOP code can be kept in a separate location and the OO model can remain intact. The AOP code is woven with the application.

Below are a few concepts related to AOP:

- **Advice**: The advice represents the additional code to be applied to the existing model.
- **Point-cut**: The point-cut is the point of execution in the application at

which a distinctive behavior “cuts” across multiple points in the OO model. For example, a point-cut is reached when the thread enters a method.

- **Aspect**: The aspect is composed of the point-cut and the advice.

Chapter 3

OVERVIEW AND MOTIVATION

Developing GUI testing tools presents a set of several challenges. First, an adequate representation of the GUI should be created and used across these tools. The GUI should be represented at a high level of abstraction that would overcome non functional discrepancies. Second, GUIs respond to user-generated events [21].

To test GUIs, we need to simulate these events then check the result of the action. Driving the execution of the GUI is not easy and could be prone to error. Some of the widgets could be skipped during the process which requires human intervention to make sure that the GUI is being adequately tested. Therefore tests should be automated as much as possible to simplify the test designer's work.

Additionally, conventional test coverage criteria might not catch all the user interaction scenarios. The amount of code being tested is important, but what's equally important is whether the tested code corresponds to problematic user interactions [22].

On the other hand, a non crashing application does not implicate a correct

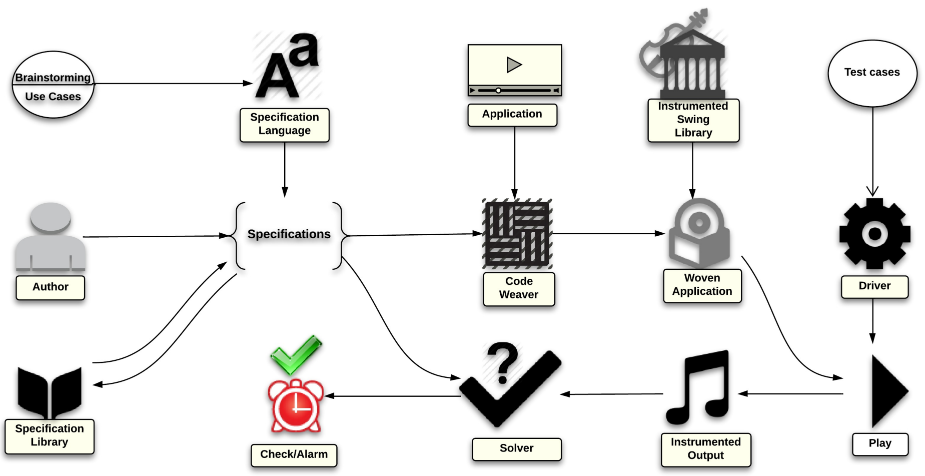


Figure 3.1: GUICop Flow Diagram.

GUI. Some of the elements might not be appearing or they might have a ”bad look”. The appearance of the GUI should be equally critical to its functionality and detecting problems in the layout cannot be achieved by only exercising GUI events.

In our approach illustrated in Figure 3.1, the user writes specifications to describe the GUI under test using the GUICop *specification language*. The *specification language* was designed after brainstorming and producing around 50 *use cases*. The specification represents therefore the screen output that the tester expects to see.

A *specification library* maintains the models (primitive or complex shapes) defined in the user's specification which allows for their reuse in the future.

GUICop checks the specification against the GUI without modifying the application's source code. A *code weaver* weaves the call for GUICop with the application to intiate the checking process.

```
private JButton findButtonWithText(String text) {
  for (Component c : getComponents()) {
    if (c instanceof JButton && c.parent() instanceof JFrame) {
      JButton button = (JButton) c;
        if (text.equals(button.getText())) return button;
        }
    }
return null;
}
...
findButtonWithText("Method");
```

Figure 3.2: GUI correctness via programming.

To capture the behavior of the GUI, the underlying graphics library is instrumented to generate a list of the shapes being painted on the screen. This output is derived from the *instrumented library* by means of *driving* the execution of the GUI through a *driver*.

Finally, the exhibited behavior is checked against the specification using our *solver* and the check either *passes* or *fails*.

3.1 GUICop vs. GUI Correctness via programming

We now present an example showing the advantages of GUICop over the common approach of programmatically checking for GUI correctness.

Given the associated requirement: *A dialog box has a button containing the text "Method"*. The above could be checked programmatically using the specification shown in Figure 3.2.

The code searches for the button among the GUI widgets via the class `JButton`. It also looks for the parent of the button by checking if it's a frame (instance of `JFrame`). Once the button is found, the text inside the button is matched against the text 'Method'.

We can foresee the following problematic points in this code:

1. Writing this code requires that the tester has enough expertise on how the GUI components are represented in the GUI tree.

2. This code needs to be tailored/ported for each supported GUI API such as Qt, MFC and SWT.

3. What if the text inside the button is cropped or not visible?

Using GUICop, the alternative specification would be as shown in Figure 3.3

```
DialogBox = {
  variables {
    rectangle r1, r2;
    textrect t1;
  }
  properties {
    X = r1.x;
    Y = r1.y;
    WIDTH = r1.width ;
    HEIGHT = r1.height ;
  }
  constraints {
    (((r1 contains r2) and (r2 contain t1))
    and (t1.string == 'Method'));
  }
}
```

Figure 3.3: Specification as done using GUICop

In this specification, we can note the following:

1. The tester does not need to know that a GUI tree exists.

2. The specification portability is not an issue due to the high abstraction level the specification is written at.

3. The issue of component visibility is implicitly taken care of by the instrumented GUI libraries within GUICop.

```
hscrollbar = {
   variables {
     triangle  tr1, tr2;
     rectangle r1, r2, r3, r4;
   }

   properties {
                X = r1.x;
                Y = r1.y;
                WIDTH = r1.width+r2.width+r4.width;
                HEIGHT = r1.height;
        }

   constraints {
   (((r1 contains tr1) leftto (r2 contains r3)) leftto (r4 contains tr2));
 }
}
```

Figure 3.4: Specification of a horizontal scrollbar.

3.2 GUICop in Action

We now present an example to show how GUICop can check a specification against an instrumentation output and return whether the specification is met or violated.

The following specification defines a horizontal scrollbar `hscrollbar`, shown in Figure 3.4. The specification has one `specobject` to be checked: `hscrollbar`. The first section of the specification is the declaration of the variables. The variables declared are two triangles: tr_1 and tr_2 and four rectangles: r_1, r_2, r_3 and r_4.

The second section assigns the properties of the horizontal scrollbar. X, Y, and HEIGHT correspond to x, y and height of rectangle r_1. The width of the horizontal scrollbar is taken to be the summation of the widths of r_1, r_2 and r_4, i.e. the bordering rectangles.

The third section states the consraint that dictates how the variables are positioned with respect to each other. Rectangle r_1 contains triangle tr_1. Rectangle r_2 contains rectangle r_3 and rectangle r_4 contains triangle tr_2. r_3 represents the knob of the horizontal scrollbar.

```
rectangle(10, 10, 100, 10); //x, y, width, height

polyline(4, 12, 15, //triangle: # of points,x1,y1
    ,
20, 18, 20, 12, 12, 15); //x2,y2,x3,y3,x1,y1

polyline(4, 108, 15,
100, 18, 100, 12, 108, 15); // triangle
rectangle(60, 12, 6, 6); // knob
...
rectangle(72, 12, 6, 6); // knob moved right
...
rectangle(78, 12, 6, 6); // knob moved again
```

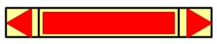


Figure 3.5: Sample instrumentation output(left); horizontal scrollbar(right).

Figure 3.5(left) shows a sample instrumentation output that draws a horizontal scrollbar and does not violate the specification. Figure 3.5(right) shows the horizontal scrollbar. The comments are not part of the instrumentation output and are added for clarification. In case the knob moves beyond the right triangle as follows, the checker determines that the constraints are violated.

```
rectangle(103,12,6,6);//(knob leftof t2) violated
```

Chapter 4

RELATED WORK

In this section, we provide an overview on the different tools and approaches relevant to the work we developed. In the last decade, several GUI testing tools have been developed to be used by the academia or the industry.

Many of the existing GUI testing tools are script-based, i.e. they require the user to manually write unit tests to validate the behavior of the GUI application in order to automate the test execution, such as Abbot[1], Pounder[2], JFCUnit[3], SWTBot[4], UISpec4J[5], Selenium WebDriver[6], Robotium[7] and SOAtest[8]. Testers write test scripts to interact with the GUI, i.e by invoking clicks or keyboard strokes, to perform interaction during playback. Test cases validate whether the application executed correctly.

Abbot [23] is an extension of the JUnit framework that supports writing specifications for programmable Java GUI components. Testing with Abbot

[1]abbot.sourceforge.net
[2]pounder.sourceforge.net
[3]jfcunit.sourceforge.net
[4]swtbot.org
[5]www.uispec4j.org
[6]www.seleniumhq.org/projects/webdriver
[7]robotium.com
[8]https://www.parasoft.com/product/soatest

consists of writing scripts to get references to GUI components and either perform user actions on those components or make some assertions about their state [23].

Writing such scripts is tedious and prone to error since there isn't a systematic or general way to follow. Abbot also stops short of enabling the user to specify general layout for the GUI. For example, a component may match its programmable specification, even if it was partially hidden by another component on the screen. The same applies to other JUnit extensions used to write assertions such as JFCUnit, Pounder, Marathon, SWTBot, UISpec4J, and Jemmy.

Similar script-based tools have been developed based on visual GUI testing. Skiuli[9] for instance is one of the tools that use computer vision techniques to implement a visual language for writing scripts. The testers can use screenshots of GUI elements to visually find them instead of using the names or properties of the components [3] [2]. For example, a tester can write the following script: `click(>); assertExist(||); assertNotExist(>);`. This script states that when the play button is pressed, a pause button should automatically replace it. '`>`' and '`||`' refer to real snapshots in the Sikuli environment.

Unfortunately, Sikuli offers a method that is associated with robustness problems and is highly dependent on computationally expensive image processing techniques to abstract the strict comparisons. In short, script-based testing can reduce the effort put in GUI testing since test cases can be used to automate GUI interactions. However, writing these scripts is a very time consuming process, therefore the tester may not be able to develop as many test cases.

In contrast with script-based testing, other techniques called Capture/Replay tools would capture the user sessions and replay them later, such as jrapture, testworks, Test Automation FX, Selenium IDE, Rational Robot, HP WinRunner

[9]

and Quick Test Pro.

Capture/Replay tools are less labor intensive than script-based tools since they involve more interaction; however the tester still has to manually perform the capturing process even though the test cases can be automatically replayed for regression testing [24] [25]. This technique suffers from lack of robustness to GUI changes. Both techniques lead to a relatively small number of test cases since they lack enough automation.

Another generation of GUI testing tools operate by directly accessing the structure of the GUI through widgets, widget properties and values which makes such tools more robust to screen resolution and GUI changes than the previous ones [26]. GUITAR [11] is an example that incorporates a GUI ripping process [5] and an automated test case generation.

Memon et al. worked considerably on developing GUI Ripper [5], a technique for dynamically reverse engineering models of GUI applications for the purpose of automating the testing process [5] [27] [28] [29]. GUITAR [11], a tool developed by Memon and al. is a model based system for automated GUI testing. The GUI ripper is part of the GUITAR toolset. It works on extracting the structure of the GUI by invoking all windows and executing all executable widgets in a window.

The GUI structure is represented by a GUI forest where every node represents a window that captures the widgets, properties and values in that window [5]. GUITAR then proceeds to converting the GUI forest into an event flow graph [11] which is then passed as an input for a test case generator to create test sequences. The generated test sequences can therefore be executed automatically by being replayed.

Unfortunately GUITAR falls short of being able to model dynamically changing GUIs [30]. Parts of the GUI might be missing from the generated test cases [31], for example when a GUI component is partially visible due to another component. GUITAR performs GUI regression testing: it considers the given GUI as a reference to produce test cases. If the GUI has defects, GUITAR generates tests that consider these defects as a correct behavior [32].

To overcome this issue, the testing process should be based on specifications of the GUI, i.e. what we expect the GUI to look like. This is where our approach comes in. The tester using GUICop separately defines a set of specifications that presents a high level of abstraction of the GUI layout. The GUI layout is simplified by primitive and/or custom defined objects and a description of the relationship between the objects. GUICop therefore does not rely on visual testing and therefore shows robustness against problems due to image and screen resolution.

The GUICop toolset has a driver which is loosely based on the GUI Ripper to drive (automatically) the execution of the GUI application. The GUI Ripper extracts the structure of the GUI and generates a GUI forest where each node represents a window and captures the widgets, properties and values in that window. GUICop on the other hand focuses on an instrumented version of the graphics library to capture the behavior of the application GUI. This improves the ability of GUICop to detect defects associated with the display that are undetectable in the internal properties of the widgets maintained by the GUI forest.

Chapter 5

SPECIFICATION LANGUAGE

GUICop takes a specification that describes the desired behavior of the GUI. The specification is written in a GUICop specification language designed to cover positional, arithmetic and logical GUI behaviors. To design this language, we assembled around 50 case scenarios that we expressed in English first. Then we designed a hierarchical specification language with primitive objects such as `Rectangle`, `Ellipse` and `Line` that can be directly instantiated with keywords. More complex objects can be declared with hierarchical composition.

The language allows the following.

1. **Object Declaration**

   The first step in writing a specification is to declare the variables to be used. Variables are the objects used to construct the `specobject`, the object we're checking for. These variables can represent primitive objects such as `Triangle`, `Rectangle`, `Ellipse`, `Polygon`, `Textrect` and `Line` or they can be complex objects declared by the user. A variable declaration includes the type of the object followed by an identifier, such as *rectangle* r_1.

2. **Object Properties**

The properties of an object are the key to compute positional, logical and arithmetic checks in the specification. Every primitive object has its set of properties. These are listed in Chapter 6. The properties of complex objects or `specobjects` are modeled by a bounding box rectangle. They are therefore labeled as X, Y, WIDTH and HEIGHT. This allows for a hierarchical composition where a `specobject` can be used in other specifications.

3. **Object Constraints**

   A specification describes how primitive or complex objects appear and how they're positioned with respect to each other. A constraint is an expression of operations that use objects as operands with positional, arithmetic or logical operators to express corresponding GUI behaviors.

4. **Hierarchical Object Type Composition**

   A `specobject`, once defined in a specification, can be used in the declaration of more complex objects. In that sense, several specifications can be assembled in one specification file.

5. **Operators**

   Different types of operations can be covered in a specification. Positional, arithmetic and logical operators are therefore used in a constraint to check for a specific GUI behavior. These operators take two operands which can be:

   - ID of primitive objects or complex objects when checking for a positional GUI behavior.

   - access for a member variable, i.e.$r_1.width$ when computing an arithmetic operation.

Binary operators are divided as follows:

- logic such as *and, or,* $==$ and $<$
- positional such as *above, contains, smaller, equal, leftto*, and *over*
- alignment such as *leftaligned, rightaligned, topaligned*, and *bottomaligned*
- arithmetic such as $+, -, *$, and $/$

5.1 Use Case Scenarios

To be able to define the specification language, we inspected around 50 case scenarios that we expressed in English first. The result of these experiments and several brainstorming sessions led to designing the specification language in the way we explained before. The case scenarios include the main interfaces of standalone applications such as *Calculator* and *Music Player*, or specific parts of the GUI such as *Menu Bar*, *Top Bar*, *Scrollbar* and *Drop Down List.*

A specification example for *Drop Down List* when it's collapsed is shown in Figure 5.1:

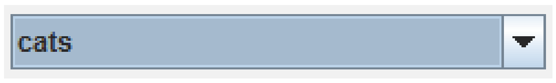


Figure 5.1: Collapsed Drop Down List

The user is checking for a rectangle containing two other rectangles. One of them contains the text 'cats', the other one contains a downwards triangle. The specification in this case scenario was only an expression of constraints. The

```
Drop_Down_List =$rectangle(r1) CONTAINS (rectangle(r2) CONTAINS text(t1) LEFTTO
    Expand_Arrow) AND (t1.value = 'cats')
Expand_Arrow = rectangle(r1) CONTAINS triangle(t1) AND (t1.x1 < t1.x2) AND (t1.
    y1 = t1.y2) AND (t1.x3 > t1.x1) AND (t1.x3 < t1.x2)
```

Figure 5.2: An early stage specification of a collapsed drop down menu.

objects were declared within the expression itself and the properties weren't cached after the check is complete so we could not use the `specobject` in further specifications.

Realizing how such a specification was incomplete and rather complicated to build a checker accordingly, we decided to divide the specification into three sections, a section for the declaration of variables, a section for caching the properties and a section for expressing constraints.

5.2 Grammar

The grammar in Figure 5.3 describes the specification language and it is written using ANTLR. An adequate overview about ANTLR can be found in Chapter 2.

The grammar is composed of a set of rules. Any rule that can be defined using tokens (concatenated characters or repetition of characters) is called a regular set. Any rule that can be defined in terms of tokens or recursion is called a context-free language. Tokens are repeated using regular expressions. The regular expression $*$ used in the grammar means 0 or more repeated tokens. The output of this grammar is an Abstract Syntax Tree (AST). An AST is a simplified syntactic representation of the source code. Syntactic nodes that aren't useful for analysis such as **;**, **,** , =, {, }, **(** and **)** are excluded from the AST using **!**. The tree has operators as internal nodes and operands as leaves. To apply this structure, an operator in a rule is forced to be the root of the corresponding subtree, using the

```
grammar spec;
specobjects: specobject (specobject)*;

specobject: (ID^ '='! '{'! variables  properties constraints '}'!);

variables: ('variables'^ '{'!  variablesdecl* '}'!);

properties: ('properties'^ '{'! propertiesdecl* '}'!);

constraints: ('constraints'^ '{'! constraintsdecl ';'! '}'!);

variablesdecl: (ID^ ID (','! ID)*';'!);

propertiesdecl: (PROPERTY^ '='! expression ';'!);

constraintsdecl: '('! constraintsdecl OPERATOR^ constraintsdecl ')'!
     | membervariableaccess | ID | INT | QUOTEDSTRING;

expression: membervariableaccess (OPERATOR^ membervariableaccess)*;

membervariableaccess: (ID '.'^ ID);

OPERATOR: 'leftto' | 'rightto' | 'above' | 'below' | 'contains'
        | 'over' | 'smaller' | 'leftaligned' | 'rightaligned'
        | 'topaligned' | 'bottomaligned' | 'textsubstring'
        | 'textsmaller' | 'textconcatenate' | 'and' | 'or'
        | 'xor' | '+' | '-' | '*' | '/'  | '==' | '.'
        | '<' | '>' | '!=' ;

PROPERTY: 'X' | 'Y' | 'WIDTH' | 'HEIGHT';
```

Figure 5.3: Grammar of the specification language

symbol ^.

The rules that fall under the regular sets are OPERATOR and PROPERTY. OPERATOR denotes the keywords used for positional, arithmetic and logic operators. PROPERTY denotes the properties of the bounding box rectangle generated after the check is complete and satisfied: X, Y, WIDTH and HEIGHT.

The rules that fall under the context-free language are membervariableaccess, variablesdecl, expression, propertiesdecl, constraintsdecl, specobjects, specobject, variables, properties and constraints. The rules are explained below:

- **specobject**: this rule represents the specification of a specobject. A specobject is defined using its ID followed by the list of declared variables, the list of properties and the constraints expression.

- **specobjects**: the specification language allows for a hierarchical composition. This rule denotes the definition of at least one specobject in the file, with the possibility of having 0 or more other specifications if a specobject is defined using other specobjects.

- **variablesdecl**: this rule denotes the declaration of the member objects. A declaration includes the type of the object followed by one or more identifiers.

- **variables**: this rule denotes the section of variables. It calls the rule variablesdecl which declares the member objects. The rule Variablesdecl can be repeated more than once in order to declare several types of member objects.

- **membervariableaccess**: this rule denotes access to properties of a member object.

- **expression**: this rule denotes one or more operations with access to properties of member objects as the operands.

- **propertiesdecl**: this rule declares the properties of the specobject. X, Y, WIDTH and HEIGHT are defined in terms of properties of member objects.

- **properties**: this rule declares the section of properties.

- **constraintsdecl**: this rule denotes the constraint on how member objects should appear in the GUI. This rule is recursive. The constraint is an expression of operations with member objects or properties of member objects as operands. The base case is any ID of a member object, string, integer or properties of member objects.

- **constraints**: this rule declares the section of constraints.

5.3 Abstract Syntax Tree and Parse Tree

The parser in ANTLR constructs a Parse Tree and returns an Abstract Syntax Tree representing the grammatical structure of the parsed input. An AST represents semantically meaningful aspects of the input program, in contrast with a parse tree which shows the concrete syntax, i.e. how tokens are grouped together and records the complete textual form of the input. Certain keywords and punctuation such as ;, , , =, {, } (and) are omitted from the AST since the compiler only cares about the abstract structure. However they are included in a parse tree where the grammar derivation is fully represented.

GUICOP is provided with the Abstract Syntax Tree generated by the parser. The solver traverses the AST in order to verify that the instrumentation output meets the specification assigned by the user. Every `specobject` in an AST is a tree holding a subtree of variables, a subtree of properties and a subtree of constraints. The root node of the AST is an empty node with the defined `specobjects` as its children.

The data structure of the AST has the following:

1. The root node has an empty node
2. The children of the root node are the trees for the defined objects in a given output file
3. Each object node has 3 children:

```
pushedradiobutton = {
        variables {
                ellipse e1, e2;
        }
        properties {
                X = e1.x;
                Y = e1.y;
                WIDTH = e1.width;
                HEIGHT = e1.height;
        }
        constraints {
                (e1 contains e2);
        }
}
```

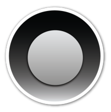


Figure 5.4: Pushed radio button specification (left); (right) Pushed radio button

- variables tree, subtree that contains a list of member objects that can be primitive or other `specobjects`.
- properties tree, subtree of 4 nodes labeled X, Y, WIDTH and HEIGHT of the bounding box object.
- constraints tree, a binary tree that has the operators as inner nodes and the declared member objects as leaves.

5.4 Example

We now present an example of a specification written by a user to check for a pushed radio button. In the normal state, the shape of a radio button is an ellipse. When it is pressed, it appears as two ellipses, one inside the other. Figure 5.4(left) shows the pushed radio button specification. Figure 5.4(right) shows the picture of a pushed radio button.

In the first section of the specification, the objects that construct the radio button are declared. The declaration includes the type of the object *ellipse* followed by two identifiers e_1 and e_2. The second section lists the properties of the radio button. These properties take after the properties of the container ellipse e_1. In the last section, the constraint states that ellipse e_2 is contained inside ellipse e_1.

5.4.1 Abstract Syntax Tree

The Abstract Syntax Tree as generated by the parser in ANTLR is shown in Figure 5.5. The tree structure conforms with the one described in section 5.3 and only keeps the meaningful representation of the grammar that is needed by the compiler.

The root node is an empty node `nil`. The root node has the `specobjects` defined in the specification file as its children. In this case, there's only one child corresponding to one `specobject`, the radio button. The `radiobutton` node has three child subtrees: variables, properties and constraints.

The variables tree has the types of declared member objects as children of the `variables` node, such as `ellipse` in our example. The identifiers `e1` and `e2` are the children of the type node `ellipse`.

The properties tree has X, Y, WIDTH and HEIGHT as children of the `properties` node. This is common across every `specobject`. Every property node has either the property of a declared member object as a child, or an expression (operation) between properties of declared member objects. In our example, the properties of the radio button take after the properties of the ellipse e_1. The dot '.' operator is the child of every property node; it goes into the internal nodes to denote access to a property of a member object. The identifier of the member object e_1 and its properties are cached in the leaves.

The constraints tree has the expression of the constraint as a child of the `constraints` node. Operators are stored in internal nodes and operands are stored in leaves: `e1` and `e2` go into the leaves, the operator `contains` goes into their parent node.

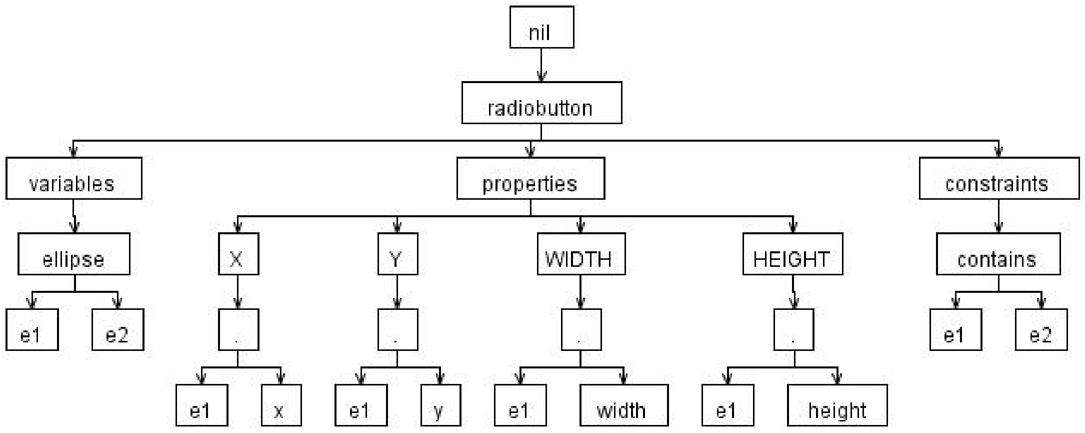


Figure 5.5: Abstract Syntax Tree

The Abstract Syntax Tree is the key to semantically analyze the specification input. GUICop generates using ANTLR the corresponding AST of the specification and extracts information into data structures. The solver then augments the tree structure of the constraints and recursively traverses the tree in order to compute the check.

5.4.2 Parse Tree

The Parse Tree of the variables section constructed by the parser in ANTLR is shown in Figure 5.6. A full parse tree for the specification above is very large and cannot be scaled to fit into the page. The parse tree graphically represents the derivation sequence of the input. The tree nodes represent symbols of the grammar. There's one leaf for every token and one internal node for every reduction during parsingPunctuation and keywords such as ;, , , =, {, } (and) are all included.

The root of the tree is the title of the grammar, `grammar spec`. The root has only one element `variables` as a child, since we're only interested in showing the grammar derivation of the variables rule.

The variables rule has a string, another rule and keywords such as {, } and WHITE SPACE as children. The string is the keyword `variables` that denotes the section, followed by the `variablesdecl` rule. The `variablesdecl` rule is a node having the type of the member object `ellipse` followed by the identifiers `e1` and `e2` as children. WHITE SPACE, **,** and **;** also appear as children of the `variablesdecl` node.

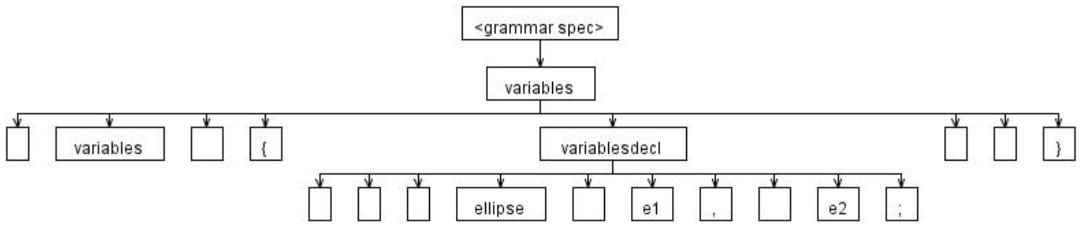


Figure 5.6: Parse Tree of the variables section

The Parse Tree is the key to verify that the written specification is syntactically correct. GUICop parses the specification input using ANTLR. If the specification written by the user doesn't conform with the grammar rules, GUICop raises an error to alert the user.

Chapter 6

INSTRUMENTATION

To capture the behavior of the GUI, GUICop saves the list of components being drawn on the screen in an output file. The components are generated by the graphics library used to build GUIs. The components available in the list are then matched against the primitive objects defined in the specification.

Qt and Java Swing are widely used toolkits for building GUI applications for the C++ and Java platforms, respectively. Supporting them within *GUICop* requires instrumenting their APIs to capture the behavior of the rendered GUIs.

The instrumentation is mainly performed on functions that draw basic shapes such as rectangle,line and text. The process includes adding printing statements to these functions. An application executed using the instrumented libraries generates a GUI Output file containing the list of all the basic shapes that are drawn on the screen with their properties. Currently, the shapes that are supported in an output file are shown in Figure 6.1: Qt, a library written in C++, was chosen to be instrumented in the final year project. For this project, the choice was expanded to explore a different platform than C++ which led us to Swing, the next-generation GUI toolkit that Sun Microsystems created to enable

```
rectangle(x, y, w, h);
line(x1, x2, y1, y2);
oval(x, y, w, h);
polygon(x1, y1, ..., xn,yn);
triangle(x1, y1, x2, y2, x3, y3);
text(x, y, str);
textrect(x, y, w, h, str);
```

Figure 6.1: Primitive shapes and their properties

enterprise development in Java [16].

Before settling on Swing, SWT (Standard Widget Toolkit) which is also a graphical widget toolkit for use with the Java platform, was an option. According to Eclipse Foundation, SWT and Swing were built with different goals in mind. SWT is oriented towards high performance, native look and feel and deep platform integration. Native widgets can be accessed through SWT across a spectrum of platforms. Swing, on the other hand, is designed to allow for a highly customizable look and feel that is common across all platforms [33].

Both have rich component types and features; however Swing is a standard GUI library that gets shipped with JRE, there is no need for additional native libraries which made it very desirable especially that it also presents a high level ease of use. A brief overview about Java Swing is available in Chapter 2.

The output file generated by the instrumented graphics library follows a specific format that corresponds to an output language we designed using ANTLR. The parser in ANTLR returns an Abstract Syntax Tree (AST) that is passed to GUICop. GUICop reads the AST and saves the objects in a list of strings.

6.1 Java 2D API

The Java 2D API offers advanced 2D graphics and imaging through a set of classes that enable the development of richer Java applications and user interfaces. The classes are provided as enhancement to the Abstract Windowing Toolkit (AWT) [18].

6.1.1 Related Packages

Swing draws the objects on the screen using the Java 2D API. The API has several packages; three of them were instrumented to produce a list of all the objects drawn on the screen. The packages that are involved in the instrumentation are described as follows:

- `javax.swing.plaf.metal`: Metal is the default Java look and feel for Swing components. It is shipped with Swing and is not related to a specific platform. This is the crossplatform and the most general look-and-feel that will be used if one does nothing in the code to set a different L&F [34]. MetalIconFactory.java is the instrumented class in this package. It enables creating radio buttons and check boxes.

- `sun.java2d`: The Java 2D expands AWT to support more general graphics and rendering operations. The Graphics class is behing drawing all the primitive objects such as rectangles, ovals, polygons and lines. Graphics2D provides a way to render virtually any geometric shape [18].

- `sun.swing`: For historical reasons, some traditionally non-public implementation classes for Swing are provided in this package such as SwingUtilities2.java [13].

6.1.2 Instrumented Classes

The instrumented classes in Swing are `SunGraphics2D.java`, `MetalIconFactory.java` and `SwingUtilities2.java` exisiting in the packages `sun.java2d`, `javax.swing.plaf.`

```
drawRect(int x, int y, int w, int h)
drawRoundRect(int x, int y, int w, int h, int arcW, int arcH)
drawOval(int x, int y, int w, int h)
drawArc(int x, int y, int w, int h, int startAngl, int arcAngl)
drawLine(int x1, int y1, int x2, int y2) drawString(String str, int x, int y)
drawChars(char data[], int offset, int length, int x, int y)}
```

Figure 6.2: Instrumented drawing functions.

`metal` and `sun.swing` respectively.

- `SunGraphics2D.java`: SunGraphics2D.java extends Graphics2D.java and implements several functions for drawing different GUI components and shapes. Some of these functions draw the outlines of the shapes while others fill the interior of the shapes. Functions shown in Figure 6.2 were modified by adding printing statements to print the shape they're drawing and its properties to an output file.

- `MetalIconFactory.java`: Radio buttons and check boxes constitute a special case. Although a radio button looks like an oval and a check box looks like a square, however these are not drawn using the methods to draw ovals or rectangles. Instead, the corresponding function is `paintOceanIcon(Component c, Graphics g, int x, int y)` where several calls to `drawLine(int x1, int y1, int x2, int y2)` are made to build the shape of a radio button and a check box using individual lines connected to each other. For instance, the following calls shown in Figure 6.3 for the `drawLine(int x1, int y1, int x2, int y2)` function with the below values for `(x1, y1, x2, y2)` draw the oval behind the radio button.

- `SwingUtilities2.java`: A collection of utility methods for Swing. This class provides the width and height of the drawn strings and characters from the FontMetrics class of the passed in Graphics.

```
g.drawLine(4, 0, 7, 0);
g.drawLine(8, 1, 9, 1);
g.drawLine(10, 2, 10, 3);
g.drawLine(11, 4, 11, 7);
g.drawLine(10, 8, 10, 9);
g.drawLine(9, 10, 8, 10);
g.drawLine(7, 11, 4, 11);
g.drawLine(3, 10, 2, 10);
g.drawLine(1, 9, 1, 8);
g.drawLine(0, 7, 0, 4);
g.drawLine(1, 3, 1, 2);
g.drawLine(2, 1, 3, 1);
```

Figure 6.3: Drawing a radio button using the drawLine function

6.2 Grammar

The grammar in Figure 6.4 describes the output language and it is written using ANTLR. The primitive shapes are defined with their corresponding properties such as coordinates and text. The grammar parses the output file and builds an Abstract Syntax Tree. The grammar is annotated in a way to make each declared shape the root of its own tree, having its properties (coordinates, text) as children. As it is the case with the specification language, all grammar tokens such as brackets, white-space and punctuation marks do not appear in the tree.

```
grammar out;
properties
        :('rectangle'^ '('! INT ','! INT ','! INT ','! INT ')'! ';'!)
        |('line'^ '('! INT ','! INT ','! INT ','! INT ')'! ';'!)
        |('ellipse'^ '('! INT ','! INT ','! INT ','! INT ')'! ';'!)
        |('polygon'^ '('! INT ','! INT (','! INT ','! INT)* ')'! ';'!)
        |('triangle'^ '('! INT ','! INT ','! INT ','! INT ','! INT ','!
         INT ')'! ';'!)
        |('arc'^ '('! INT ','! INT ','! INT ','! INT ','! INT ','! INT ')'!
         ';'!)
        |('text'^ '('! INT ','! INT ','!  DEF ')'! ';'!)
        |('textrect'^ '('! INT ','! INT ','! INT ','! INT ','!  DEF ')'!
        ';'!)
        |('point'^ '('! INT ','! INT ')'! ';'!)
        ;
listofproperties
        :        (properties)*;
```

Figure 6.4: Grammar of the output language

The rules properties and listofproperties both fall under the context-free language. The rule properties represents the possible shapes that can be drawn by the graphics library with their properites such as coordinates and text. This rule can represent one shape with its properties at a time. The rule listofproperties calls the rule properties 0 or more times to denote a list of shapes with their properties.

6.3 Abstract Syntax Tree

Similar to the specification language, the parser in ANTLR constructs a Parse Tree and returns an Abstract Syntax Tree that GUICop is provided with. The data structure of the AST has the following:

1. The root node has an empty node

2. The children of the root node are the trees for the different shapes in the output file
3. Each shape node has as many children as its coordinates and other properties. A shape tree could be as follows:
   - the root is the name of the shape such as rectangle.
   - the children are its coordinates such as x, y, width and height.

For example, we assume for simplicity that the following two shapes, a rectangle and a line, are found in the output file:

```
rectangle(10,15,10,20);
line(2,5,8,12);
```

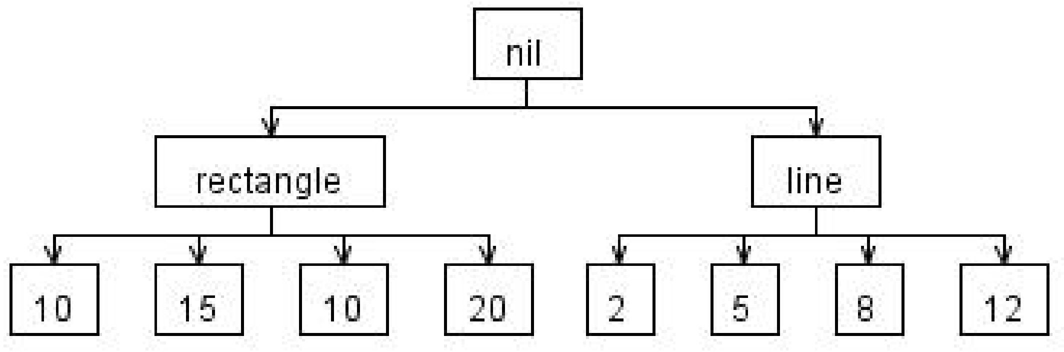


Figure 6.5: Abstract Syntax Tree

The AST generated by the parser in ANTLR shown in Figure 6.5 conforms with the structure explained above. The root node is an empty node. The children of the root node are the names of the shapes available in the output file. Every shape is a subtree with the root node being the name, and the children of the root being the properties such as coordinates and text. GUICop takes the AST, traverses it and constructs a linear list of strings with the shapes and their properties.

6.4 Special Cases

GUI applications are often refreshed generating several instances of the same objects. The list returned by the instrumentation process might therefore have many entries for the same shape and properties. For instance when we minimize a window then maximize it, all widgets are repainted again which generates several redundancies. GUICop is embedded with the ability to detect these redundancies and filter them out of the output file by tracking the stack traces for all the results and excluding outliers from the output. Below is a list of all the conditions that GUICop handles in order to omit undesired results:

- **Double Buffering**: double buffering mode is by default enabled in Swing. It is primarily used to provide a better perceived performance and user experience by avoiding to draw directly to a visible component and drawing instead to an offscreen image first [34]. This can save the application from appearing sluggish or amateurish [35]. However this ends up generating additional shapes with properties not related to the component in hand.

- **Volatile Image**: the volatile image is associated with the double buffering mode. A volatile image allows components to take advantage of accelerated graphics hardware for extremely efficient double buffering [16].

- **Border Painting**: Swing calls the paintBorder() method to paint the borders of the component. For instnce, a button's borders are drawn as four lines delimiting the rectangle that composes it. This produces additional shapes (lines) in the output file that can be considered as false positives when computing the check.

- **Gradient Drawing**: Swing provides a method to fill shapes with color gradient patterns. The shape of the pattern is redundant and is ommitted from the output file.

- **Radio Button & Check Box**: a radio button and a check box are drawn using several lines connected to each other. The resulting shape is an ellipse in the case of a radio button and a rectangle in the case of a check box. GUICop replaces these lines in the output file with the final drawn shape.

Chapter 7

CODE WEAVER

We developed a code weaver using Aspect Oriented Programming (AOP) to weave the GUICop specification in the targeted code position of the application.

The code weaver, based on AOP, allows us to inject the call for the GUICop checker into the application without having to modify the original code. The developed AOP code is kept in a Java Aspect class separate from the applicaton.

Currently, the code weaver supports three advices, as shown below:

- **Before advice**: advice code is executed before the joinpoint method.
- **After advice**: advice code is executed after the joinpoint method.
- **Around advice**: Advice code that surrounds a joinpoint method. Around advice can perform custom behavior before and after the method invocation and can also choose whether to proceed to the joinpoint.
- **When advice**: Advice code is excuted right on the execution of the joinpoint method.

The code weaver provides a user friendly way to generate the aspect code shown in Figure 7.1. The user doesn't need to be concerned about the exact AOP

```
aspect testscript {
pointcut drawMenu() : execution(void drawMenu());
after(): drawMenu(){
guicop.check("MenuItemSeparator");
}
```

Figure 7.1: Example of the aspect code generated by the code weaver .

```
title testscript
after drawMenu()
        guicop MenuItemSeparator
```

Figure 7.2: Example of how the user writes the aspect script in English.

syntax. The user can write the script in English in a certain format; the parser of the code weaver then parses the script and returns the Java aspect code.

We show in Figure 7.1 an example of how the code weaver generates an aspect script that weaves the call for the GUICop checker into the application.

The user can write the English script shown in Figure 7.2 and pass it to the parser of the code weaver.

The code weaver fetches the necessary elements to construct the aspect code such as the `title` *testscript*, the advice (`before`), the function name (`drawMenu()` and the code to call around the function which a call for the GUICop checker. *MenuItemSeparator* is the `specobject` to be checked and it is defined in the specification.

Chapter 8

SOLVER

The solver is a key module in the GUICop methodology. It checks whether the GUI at hand meets the specification assigned by the user. The solver takes as input the Abstract Syntax Tree (AST) of the specification and the AST of the instrumentation output and returns whether the output file meets the specification.

The Abstract Syntax Trees of the specification and instrumentation output are trees automatically generated by the front end from the specification file and the instrumentation output file respectively, and they are described in Chapter 5 and 6. The instrumentation output file is the result of running the GUI with the instrumented library described in Chapter 6.

The root of the AST of the specification is the empty node `NIL`. It has several child subtree objects; each subtree corresponding to a `specobject` defined in the specification file. The root of the `specobject` subtree has three children: variables, properties and constraints. The constraints is a subtree where the internal nodes represent the operations and the leaves represent the declared member objects or access to properties of member objects. The declared member objects denote member variables in the `specobject`. They can be primitive such as `Triangle`, `Rectangle`, `Ellipse` and `Line`, or other `specobjects` declared by the user. The solver traverses the top object recursively. Once a user defined object is met, its subtree is traversed. Once a primitive object is met, its matches from the instrumented file are computed and cached. Once an operation is met, the matches of its operands are computed by a recursive traversal; then the operation is applied at the matches of the operands to compute its own matches. Once the matches of the root node of a specobject subtree are computed, the properties are computed. The properties represent the coordinates X, Y, the WIDTH and the HEIGHT of the rectangle bounding the `specobject`.

The AST of the instrumentation output has the primitive objects in the output file as children. The solver traverses the AST of the instrumentation output and stores the primitive objects in a list of strings. The solver recursively computes tuples of objects from the output file that match nodes in the AST of the specification. The recursive base case is the leave nodes: nodes that specify basic shapes. The recursion ensures that dependencies on children nodes are all resolved and computed before applying the constraints of the node in question. Every node holds the objects that satisfy the constraints of the node in the order of appearance of the objects in the definition section of the specification. It also maintains the total number of variables declared in the specification and uses a dummy object `Joker` to fill those variables that are not yet satisfied. As the tree is being traversed recursively, instances of the joker are gradually replaced by matching shapes. The solver creates a bounding box object that contains these matching shapes. At the end it will return whether the specification is met by checking that the root of the tree has at least one satisfying object.

```
hscrollbar = {
   variables {
     triangle  tr1, tr2;
     rectangle r1, r2, r3;
   }
   properties {
                  x = r1.x;
                  y = r1.y;
                  width = r1.width+r2.width+r3.width;
                  height = r1.height;
        }
   constraints {
   (((r1 contains tr1) leftto r2) leftto (r3 contains tr2));
 }
}
```

Figure 8.1: Specification of a horizontal scrollbar

8.1 Example

GUICop takes as input a user specification and an instrumented output, then returns if the output meets the specification. To illustrate the work of the solver, we take as an example a horizontal scrollbar with the specification shown in Figure 8.1. Figure 8.2(left) shows the generated instrumentation output. Figure 8.2(right) shows the GUI sample that includes a horizontal scrollbar.

```
o1: rectangle(10, 30, 10, 60);
o2: rectangle(10, 10, 10, 10);
o3: rectangle(80, 10, 10, 10);
o4: rectangle(20, 10, 60, 10);
o5: triangle(88,15,80,18,80,12);
o6: triangle(12, 15, 20, 18, 20, 12);
```

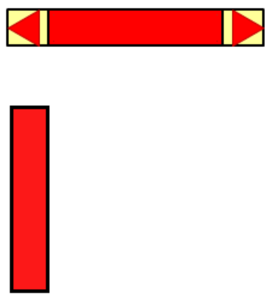


Figure 8.2: Sample instrumentation output (left) ; (right) Sample GUI including a horizontal scrollbar

The diagram in Figure 8.3 illustrates how GUICop works. The GUICop

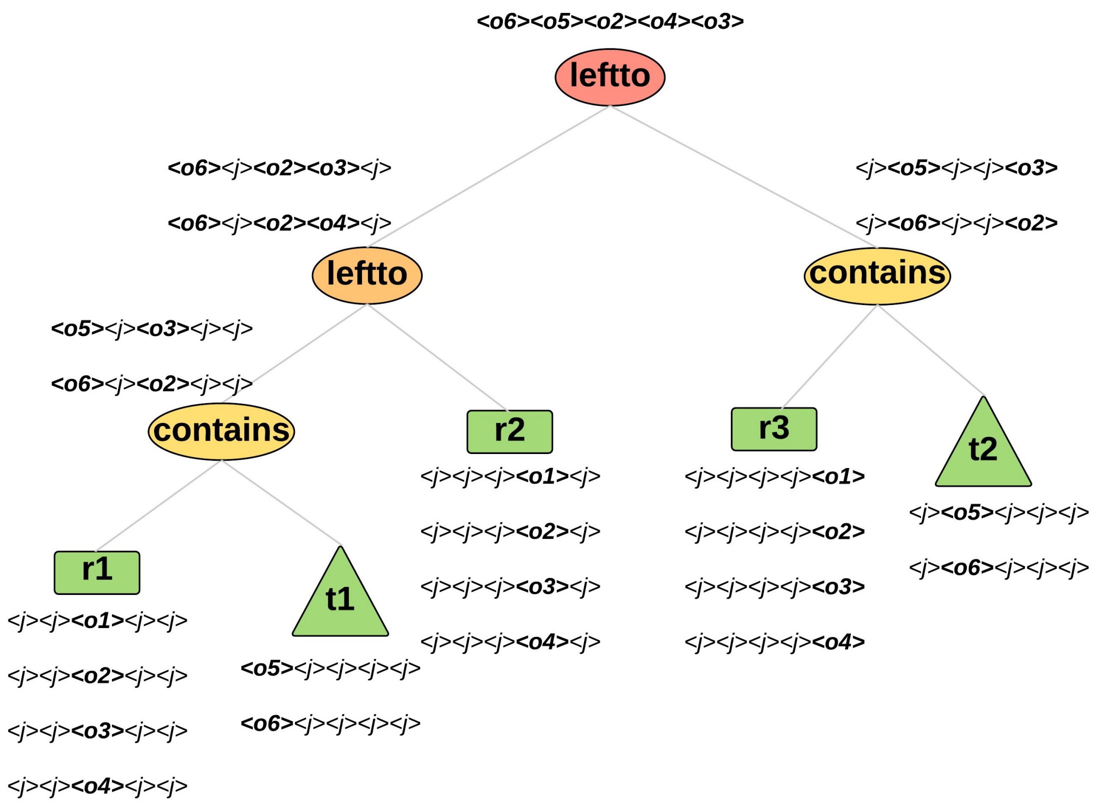


Figure 8.3: GUICop Checker

checker builds an *abstract syntax tree* (AST) from the specification and monitors the instrumentation output to check whether it satisfies the specification. The checker currently supports the `rectangle`, `triangle`, `line`, `ellipse`, `arc`, `polygon`, `text` and `textrect` shapes.

First, the checker annotates each node in the AST corresponding to a basic geometric shape or event with all the objects in the instrumentation file matching it, in addition to instances of the `Joker` denoted by $<j>$. We call each annotation a *solution*. A solution would be then a list of potential components that can satisfy the current node. The number of objects and their order is always maintained in a component. Every object appears in the component at the order it was declared in the definition section of the specification. At the leaf level of the tree, instances of the `Joker` are inserted in the place of the shapes that are not satisfied yet. As the tree is being traversed recursively with operations being solved, instances of the `Joker` are gradually replaced with shapes that satisfy the operation in question.

The tree in Figure 8.3 is the AST of the constraint of the `hscrollbar` example. Identifiers t_1 , t_2 , r_1, r_2 and r_3 are assigned to the rectangle and triangle objects from the instrumentation output. t_1 and t_2 are both triangles thus the checker annotates t_1 with $<o_5><j><j><j><j>$and $<o_6><j><j><j><j>$since t_1 appears first in the declaration of variables. t_2 comes second, therefore the checker annotates it with $<j><o_5><j><j><j>$and $<j><o_6><j><j><j>$. $<j>$a `Joker` instance replaces the shape that is not matched yet. At the level of the leaves, a typical *solution* has only one satisfied object among instances of the `Joker`.

Second, the checker recursively traverses the AST from the root node representing the specification and computes whether each node can be satisfied with the current status of the instrumentation output. Each node represents an operator and is associated with rules that satisfy it. For example, we say *o1,o2* $\vdash$ *o1 contains o2* if and only if *o1.getMostTop()* $<$*o2.getMostTop()* and *o1.getMostLeft()* $<$*o2.getMostLeft()* and *o1.getMostBottom()* $>$*o2.getMostBottom()* and *o1.getMost Right* $>$*o2.getMostRight()*. Consider the left subtree of the AST. The operator `contains` with r_1 and t_1 as operands. It has two satisfying components out of the eight possible components resulting from the Cartesian product of the respective solutions of r_1 and t_1. The checker annotates the `contains` operator with $<o_5><j><o_3><j><j>$and $<o_6><j><o_2><j><j>$as solutions. For each operator, the checker uses rules associated with the operator to compute properties for the matching pairs to be used in later checks. For example, for the `leftof` operator, we have *(o1,o2).x = min(o1.x, o2.x)*.

Similarly when the checker visits the `leftof` operator with r_2 and `contains` as operands, it annotates it with $<o_6><j><o_2><o_3><j>$and $<o_6><j><o_2><o_4>$ $<j>$as these satisfy the `leftof` operation. The same concept is applied to the right subtree of the AST.

Finally, when the checker visits the `leftof` root node, it annotates it with $<o_6><o_5><o_2><o_4><o_3>$as a final solution. The checker considers a node satisfied if it is annotated with at least one solution. Therefore, for our example, the specification is satisfied.

8.2 Algorithm

GUICop traverses the tree of constraints recursively. Each leaf in the tree contains either an Id or an access for a member variable (ex: r1.width). Two leaves are compared together using the operation specified in their parent node. Each leaf has a list of components that could potentially be a solution. A nested for loop compares each component in the first leaf with each component in the second leaf using the operator. If the operation is satisfied, components from every leaf are merged together in a new list. The new list of components is saved in a new node. The Satisfied bit of the new node will be set to true if the new list of components is non-empty, otherwise it would be set to false. The new node is then returned, replacing the parents and the leaves. The procedure is done recursively until the tree collapses. Finally the Satisfied bit of the one remaining node is returned, and it correctly states whether the specification meets the output of the instrumentation.

8.2.1 Tree Traversal

GUICop recursively traverses the constraints section to verify the object.

```
traverse(N):
if N has children then
        Left = Left(N)
        Right = Right(N)
        if Left has children then
                Left(N) = traverse(Left)
        end if
        if Right has children then
                Right(N) = traverse(Right)
        end if
        if Left is leaf and Right is leaf then
                return solve(Parent, Left, Right)
        end if
        return N
end if
```

Figure 8.4: Tree Traversal Algorithm.

8.2.2 Tree Verification

GUICop recursively checks whether each node can be satisfied.

```
solve(Parent, Left, Right):
Operation = Parent.text
List1 = Left.Components
List2 = Right.Components
        for i = 1 < Size(List1) do
                for j = 1 < Size(List2) do
                        Condition = Compare(List1(i); List2(j))
                        if Condition then
                                //Add List1(i) and List2(j) to List3
                                List3 = merge(List1(i); List2(j))
                        end if
                end for
        end for
N.Components = List3
if Size(List3) != 0 then
        N.satisfied = True
end if
return N
```

Figure 8.5: Tree Verification Algorithm

8.2.3 Merging Components

GUICop merges components together if the node is satisfied.

```
merge(C1, C2):
List1 = C1.Shapes
List2 = C2.Shapes
C3 = new Component
        for i = 1 < Size(C1) do
                if List1(i) is not a Joker and List2(i) is a Joker then
                        Add List1(i) to List3
                end if
                if List1(i) is a Joker and List2(i) is not a Joker then
                        Add List2(i) to List3
                end if
                if List1(i) is not a Joker and List2(i) is not a Joker then
                        if List1(i) = List2(i)
                                Add List1(i) to List3
                        end if
                end if
                if List1(i) is a Joker and List2(i) is a Joker then
                        Add Joker to List3
                end if
        end for
Add List3 to C3
return C3
```

Figure 8.6: Merging Components Algorithm.

8.3 Operators Semantics

The comparison function defines the semantics for the operators. Positional operators take as input two components, perform the operation using the components' bounds and return whether the operation is satisfied. These are shown in Table 8.1:

Operator	**Purpose**
`Above`	Checks if the lower bound of the first is above the upper bound of the second
`Leftto`	Checks if right bound of the first is left to the left bound of the second
`Contains`	Checks if the left and right bounds of the second are between the left and right bounds of the first, and the upper and lower bounds of the second between the upper and lower bounds of the first
`Over`	Checks if the two objects are overlapping
`LeftAligned`	Checks if the left bound of the first is aligned with that of the second
`RightAligned`	Checks if the right bound of the first is aligned with that of the second
`TopAligned`	Checks if the top bound of the first is aligned with that of the second
`BottomAligned`	Checks if the bottom bound of the first is aligned with that of the second
`Smaller`	Checks if the size of the first is smaller than the size of the second
`Equal`	Checks if the size of the first is equal to the size of the second

Table 8.1: Positional operators

To solve for logic operations, the implemented functions take two nodes as parameters and perform the operation on their SATISFIED bits. Table 8.2 lists the logic operators.

Operator	**Purpose**
`And`	Checks if both children nodes have the SATISFIED flag set to true
`Or`	Checks if either one of the children has the SATISFIED flag set to true
`Equals`	X == Y
`LessThan`	X < Y

Table 8.2: Logic operators

GUICop also supports arithmetic operations listed in Table 8.3. An operator in this case is an access to a member variable, ex: r1.height, r1.x.

Operator	**Purpose**
`Plus`	X + Y
`Minus`	X - Y
`times`	X * Y
`Over`	X / Y

Table 8.3: Arithmetic operators

Chapter 9

DRIVER

A graphical user interface is by default hierarchical where widgets are connected to each other through sequences of events. Using this notion, GUI testing can be automatically "driven" by exploiting this hierarchy. Following Memon's GUI ripping technique [5], we implemented a driver to act on any Swing application: it extracts the GUI's structure by starting with the top level windows and opening all executable child windows through a depth first traversal.

We present in Figure 9.1 the algorithm based on the GUI ripping process to drive the execution of the GUI of a Swing application.

The algorithm describes two methods `traverse` and `rec-traverse` that traverse

```
traverse(Application A)
ListW = A.Windows
        for W in ListW
                rec-traverse(W)

rec-traverse(Window W)
ListC = W.Components
        for C in ListC
                execute(C)
ListG = getinvokedWindows(C)
        for G in ListG
                rec-traverse(G)
```

Figure 9.1: Algorithm of the driver based on the GUI Ripper.

the GUI to extract its structure. The methods perform a depth-first traversal (DFS) on the hierarchical structure of the GUI application. A GUI is generally described as a forest of trees where each tree is rooted at a top frame or window. The method `traverse` gets the list of top-level windows $ListW$, i.e. the windows that are visible when the application first starts.

The method rec-traverse performs a depth-first search of the GUI tree rooted at the GUI window W available from $ListW$. It gets then the list of widgets or components $ListC$ that are child nodes of the window W. Every executable component is executed. The method is then performed recursively on the invoked window G available from the list of invoked windows $ListG$ until all the structure of the forest is exploited.

The driver can currently execute a set of widgets: buttons, lists, menu bars and menu items. It can also modify a text component (textfield, textarea) by typing inside of it.

Chapter 10

IMPLEMENTATION

10.1 Solver

The solver is developed in Java and has four packages in total:

- `geometric`: a package for the geometric shapes
- `structures`: a package for the structures
- `guicop`: a package for the solver's logic
- `parsers`: a package for the parsers generated by ANTLR

10.1.1 Package geometric

The `geometric` package contains all the geometric shapes that GUICop supports. An abstract class `Shape` is defined to inherit the geometric objects: `Rectangle`, `Ellipse`, `Line`, `Polygon`, `Text`, `Textrect`. An additional shape, `Joker` is defined to represent shapes that are not solved for yet during the checking process, in order to maintain the total number of variables and their order at every step. The `Shape` class defines the abstract functions shown below:

- `getLeftBound()`

Shape	**Properties**
`Rectangle`	`X,Y, Width, Height`
`Ellipse`	`X, Y, Width, Height`
`Line`	`X1, Y1, X2, Y2`
`Polygon`	`ArrayX, ArrayY`
`Text`	`X, Y, String`
`Textrect`	`X, Y, Width, Height, String`

Table 10.1: Properties of shapes

- `getRightBound()`
- `getTopBound()`
- `getBottomBound()`
- `getSize()`

These functions are common to all the previously listed shapes. Every primitive shape class inherits from `Shape` and implements more functions depending on the underlying shape. The functions are related to the properties of the shape. Every inherited shape also has a default constructor and a constructor that takes as arguments its corresponding properties. By default, all the properties of he `Joker` are set to 0.

We present in Table 10.1 the properties of every geometric shape.

10.1.2 Package structures

The `structures` package refers to the data and tree structures used by GUICop in the checking process. A `Component` class groups several basic shapes together inside a bounding box when a node is satisfied. `Component` maintains a list of shapes, and the following member functions: `getMostLeft()` gets the left side coordinate of the object in the list that has the left most left side. `getMostRight()`, `getMostTop()`, `getMostBottom()` are similar member functions.

A `Variable` class maps an id defined in the variables section of the GUI object definition to a list of potential shapes drawn from the list output of the drawing library. The class has the member variables `Id` and the list of shapes `Instances`.

Finally, a `Node` class represents the augmented node of the constraints tree. The parser generated by ANTLR parses the specification and returns an Abtract Syntax Tree (AST) to the GUICop checker. The GUICop checker fetches the variables subtree, the properties subtree and the constraints subtree. The AST has a `Tree` data structure defined in the ANTLR runtime library where the ANTLR Tree node can only hold a `String` member variable. Therefore the constraints subtree cannot be traversed before rebuilding its structure and augmenting the capacity of every node. The member variables added to the `Node` class are as follows:

- SATISFIED: a `boolean` value that is `true` if the instrumentation meets the specification at the node; `false` otherwise

- COMPONENTS : a list of objects of type `Component` that meet the specification at the node.

- VALUES : a list of doubles returned from an operation that meets the specification

- STRINGS: a list of strings returned from a string comparison that meets the specification

10.1.3 Package guicop

The `guicop` package contains the essential logic used by the solver to perform the checking process.

The instrumentation output has a list $\mathcal{V}$ of all the objects being painted on the screen. This list is constantly being updated upon screen refresh or any GUI action. GUICop monitors this list and updates it by filtering out redundancies and removing undesired results that are due to special cases listed under 6.4 in chapter 6. The list of active objects $\mathcal{L}$ is accessible by shape types, e.g. it supports an API to return all rectangles efficiently.

GUICop reads the specifications and generates the following.

1. GUICop keeps a map from each identifier, defined in the *variables* section, to its variable type in a hash table. The type serves well to access all satisfying objects for an identifier in $\mathcal{L}$.

2. GUICop keeps a map from each identifier in the *variables* section, to the order of its appearance in a hash table.

3. GUICop builds an AST for the formula in the *constraints* section. The leaves are identifiers and are associated with the objects in $\mathcal{L}$. ANTLR initially generates a tree with nodes that would only store texts. GUICop builds an augmented data structure and initializes the leaves: the variable type of the id in the leaf is fetched from the hash table. After fetching the type of the variable, a list of geometric shapes is stored inside a component of the corresponding node. If the identifier corresponds to a custom defined object, this object is checked first and a list of bounding boxes is returned and used if the custom object is satisfied. The components also contain instances of the `Joker` object to replace the variables that do not correspond to the node under consideration. An instance of the variable is any potential geometric shape that the variable can take.

4. GUICop keeps a table $\mathcal{T}$ of predicate rules indexed by the binary operators.

Each rule r_v for an operator v takes the solutions $\mathcal{S}_1$ and $\mathcal{S}_2$ of operands o_1 and o_2, respectively, and generates all possible solutions $\mathcal{S} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ and eliminates those in $\mathcal{S}$ for which r_v does not hold.

5. GUICop keeps a table $\mathcal{G}$ of rules that generate complex shapes to represent the matching solutions in $\mathcal{S}$ in terms of basic geometric shapes. The generative rules may compute the bounding box or the smallest polygon containing the operands. The bounding box is generated from the properties section of the defined object where X, Y, WIDTH and HEIGHT of the object are assigned. The GUICop checker annotates the operator nodes with the generated shapes as solutions.

6. GUICop considers the identifiers to match different objects unless otherwise specified. GUICop uses this fact to reduce the running time of the checker. For example, when the `leftof` operator from the `hsbar` example determines that `knob` is r_2, GUICop eliminates r_2 from the solutions of `hsbar` leaving only r_1.

10.1.4 Package Parsers

The `parsers` package contains the lexer and parser Java classes generated by ANTLR for both the specification and output language. The parser class has a method for every rule defined in the grammar. Rule references are translated to method calls and token references are translated to `match(TOKEN)` calls [19].

To utilize the developed grammars on the specification input and the instrumentation output, classes `ANTLRInputStream` and `CommonTokenStream` available in the `org.antlr.runtime` package are loaded to create a parser that feeds off the token stream. ANTLR then generates from the parser grammar an Abstract

Syntax Tree (AST) that is passed to GUICop. An AST is a simple and useful representation of the input. The parser also generates a parse tree, which is less relevant for us in this case. The parse tree represents the sequence of rule invocations used to match an input stream. It helps us verify that the input is syntactically correct.

10.2 Driver

The driver is implemented in Java and it is loosely based on the GUI ripping technique [5]. In Swing, GUI windows and widgets are instances of Java classes. The driver starts by launching the application under test using the reflection method where the `main` method of the application is invoked using `java.lang.reflect.Method.invoke()`. The Java API `java.awt.Window.get Windows()` identifies all the visible GUI windows. These are the top level windows of the application.

The driver starts recursively traversing these top level windows. The identity of every window is saved in a `HashMap` once accessed to avoid a second access during recursion.

The driver analyzes the current window and extracts its constituent widgets through the method `getComponents()` of class `Container` and `java.awt.Frames. getJMenuBar()` of class `MenuBar`. From the list of returned components, the driver checks if the current component is a button (belongs to the `AbstractButton` class) or a list (`JList`) or a text component (`JTextComponent`). The `AbstractButton` class is a superclass for `JButton`, `JRadioButton`, `JCheckBox` and `JMenuItem` among others. Therefore the driver can detect all types of buttons by checking if the superclass of the widget is the `AbstractButton`.

A click event is executed on the executable widgets, i.e. those belonging to the `AbstractButton` class family. This class provides the method `doClick()` that triggers a click event.

The driver traverses a list by going over (selecting) all its list elements through the method `setSelectedIndex(int i)`.

Finally, Swing offers major types of text components such as text fields (`JTextField` class) and plain text areas (`JTextArea`) class. The driver exercises a text component by typing a few characters inside of it after checking if it's editable.

After exhausting all the components belonging to one window, the process is recursively performed till all the windows are invoked and analyzed.

10.3 Code Weaver

The code weaver is written in Java. It is a class that takes as input a script written in English. The weaver parses this script to generate the corresponding Aspect Oriented code using AspectJ. The resulting AspectJ code is then woven with the application to embed the call for the GUICop checker within it.

The parser loops over the code to get all the environment variables and stores them in a `HashMap`. It then loops again to fetch the pointcuts, the identifiers, the function names and the code to be executed around the function. The parser finally constructs instances of a method called `PublicMethod` that are maintained in an `ArrayList` and that take as parameters the information previously fetched. The `PublicMethod` class has 3 member variables presented below:

- `function`: contains the function name in the application at which the pointcut should be defined

- `code`: contains the code to call around the function
- `timing`: contains information about when the code should be executed relatively to the function call in the application

Once the `PublicMethod` instance is constructed, the AOP code is generated with the correct AspectJ syntax.

Chapter 11

RESULTS

GUICop is successfully evaluated using real life case studies listed in Table 11.1. The table summarizes the applications used for the purpose of evaluating GUICop, their size (Lines of Code) and the issues that GUICop detected in these applications.

Case Study	Size (LOC)	Issue
JEdit	301,874 lines	Right to Left Justification in Arabic Writing Missing Menu Item Separator Missing Hot Key Indicator
Jajuk	86,144 lines	Unordered Listing of Numbered Tracks
Gason	1,741 lines	Cropped Content in Labels

Table 11.1: Summary of Case Studies

11.1 Case Study JEdit

JEdit is a Java open source text editor used for programmers. We wrote a specification to check against an existing JEdit defect. The defect relates to Right to Left justification for Arabic text. We also injected two more defects and wrote specifications to check against them. The first is a missing menu item separator and the second is a missing hot key indicator. We wove the GUICop specification in the corresponding code position in JEdit. We present below three scenarios where GUICop detected existing or injected bugs.

11.1.1 Right to Left Justification Defect

JEdit supports over 160 character encodings including UTF8 and UTF16 [36], therefore Arabic characters should be supported. However when writing in Arabic in the editing area, the characters appear left justified instead of being right justified.

In the event of writing in English for instance, the sentence would be left justified and the x coordinate of the resulting word would be equal to the x coordinate of the text editing area. In this case, a character or a word is represented by a textrect, which is a rectangle containing text. The coordinates of the textrect are shown relative to the surrounding text area, therefore $textrect_x = textarea_x = 0$.

We developed a specification to check whether an Arabic sentence appears right justified by verifying that the sentence does not start at the left side or the beginning of the editing area, i.e. that $textrect_x > 0$. The specification is shown in Figure 11.1.

```
RighttoLeft = {
        variables {
                firstline o1;
                secondline o2;
        }
        properties {
                X = o1.x;
                Y = o2.y;
                WIDTH = o1.width;
                HEIGHT = o1.height+o2.height;
        }
        constraints {
                (((o1 above o2) and (o1.x > 0)) and (o2.x > 0));
        }
}
firstline = {
        variables {
                textrect t1;
        }
        properties {
                X = t1.x;
                Y = t1.y;
                WIDTH = t1.width;
                HEIGHT = t1.height;
        }
        constraints {
                (t1.string == arabicstr1));
        }
}
secondline = {
        variables {
                textrect t1;
        }
        properties {
                X = t1.x;
                Y = t1.y;
                WIDTH = t1.width;
                HEIGHT = t1.height;
        }
        constraints {
                (t1.string == arabicstr2);
        }
}
```

Figure 11.1: Right to Left Justification Specification.

The specification has three `specobjects`. *RighttoLeft* is the `specobject` to be checked. The specification has a hierarchical composition where *RighttoLeft* is defined in terms of the objects *firstline* and *secondline*.

The first section of the specification of *RighttoLeft* consists of the declaration of the variables. Two custom defined objects are declared. *firstline* and *secondline* denote two lines written in Arabic to be checked if they're right justified. The specification refers to $arabicstr_1$ and $arabicstr_2$ where $arabicstr_1$ is 'ذهب الولد إلى المدرسة وتعلم درسا جديدا وأكل التفاحة و نجح في' and $arabicstr_2$ is 'الامتحان'.

The first and second lines are basically textrects, i.e. rectangles containing text which is an Arabic phrase in this case. With every character being typed, the textrect is updated with the new character and its width increases. GUICop looks for the textrect that has the full phrase in the instrumentation output file as dictated by the constraints of the $firstline$ and $secondline$ objects.

The constraint in $RighttoLeft$ states that the first line should be above the second line and that the lines should not be left justified, i.e. they should not start at the beginning of the text area. After running the application to test the scenario, the instrumented Swing library generates a long output list of shapes being drawn on the screen. The GUICop checker inspects the output of the instrumentation and asserts that the arabic phrases are not left justified, as shown in Figure 11.2.

```
Solving for: (((o1 above o2) and ((o1 . x)) > 0) and ((o2 . x)) > 0)
Number of logic operations: 0 x 0
Number of endNode components: 0 and Number of endNode strings: 0
----------------------------------------
object1 is not satisfied
----------------------------------------
```

Figure 11.2: Output of the checker. Case Jedit Right to Left justification.

11.1.2 Missing Menu Item Separator

We programmatically removed a menu item separator from a context menu in jEdit. We then wrote a specification to check for the existence of the menu item separator.

Figure 11.3(left) shows the menu with the separator between the "Properties" and "Parent Directory" menu items.

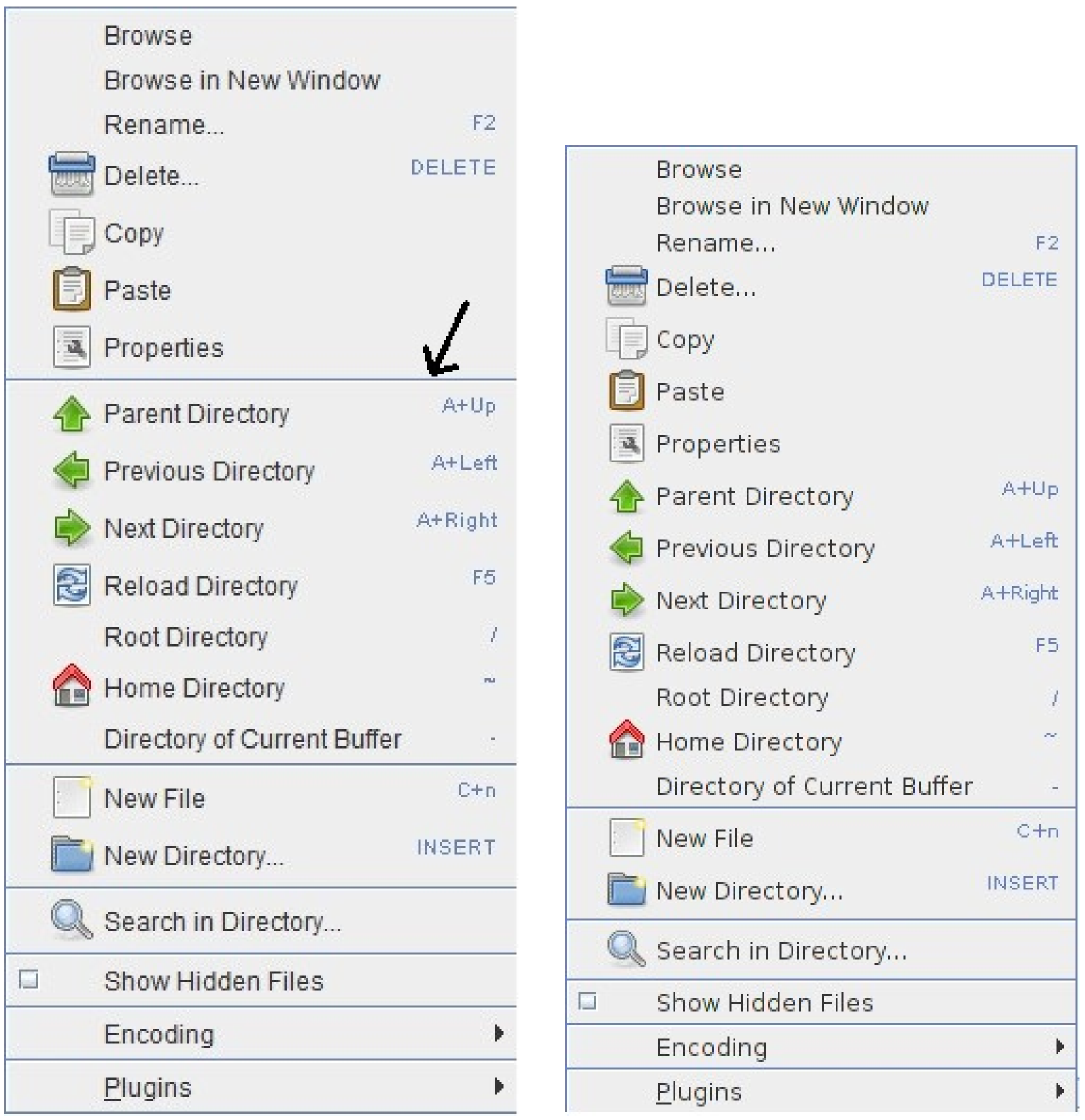


Figure 11.3: jEdit context Menu (left) with a separator between "Properties" and "Parent Directory"; (right) without the separator

Figure 11.3(right) shows how the menu looks like after rebuilding and running jEdit with the function `addSeparator()` commented. The `addSeparator()` function call in the `BrowserCommandsMenu.java` creates a separator between two menu items. Commenting this function results in the removal of the separator, as shown in the code snippet shown in Figure 11.4.

```
if ((files.length == 1) || (browser.getSelectedFiles().length != 0))
   add(createMenuItem("properties", "22x22/actions/
       document-properties.png"));
   // addSeparator();
                 }
add(createMenuItem("up", "22x22/actions/go-parent.png"));
```

Figure 11.4: Function call addSeparator() commented.

We developed the specification in Figure 11.5 to check whether the separator between the "Properties" and "Parent Directory" menu items exists. The missing separator specification has three `specobjects`.

```
MenuWithSeparator = {
        variables {
                MenuItem1 m1;
                MenuItem2 m2;
                line l1;
        }
        properties {
                X = m2.x;
                Y = m2.y;
                WIDTH = m2.width;
                HEIGHT = m2.height+m1.height;
        }
        constraints {
                ((((m2 above l1) above m1) and (l1.y1 == l1.y2)) and (l1.x1 < l1
                    .x2)));
        }
}
MenuItem1 = {
        variables {
                textrect t1;
        }
        properties {
                X = t1.x;
                Y = t1.y;
                WIDTH = t1.width;
                HEIGHT = t1.height;
        }
        constraints {
                (t1.string == 'Parent Directory');
        }
}
MenuItem2 = {
        variables {
                textrect t2;
        }
        properties {
                X = t2.x;
                Y = t2.y;
                WIDTH = t2.width;
                HEIGHT = t2.height;
        }
        constraints {
                (t2.string == 'Properties');
        }
}
```

Figure 11.5: Missing menu separator specification.

$MenuWithSeparator$ is the `specobject` to be checked. The specification has a hierarchical composition where $MenuWithSeparator$ is defined in terms of $MenuItem_1$ and $MenuItem_2$. The first section of the specification of $MenuWith Separator$ consists of the declaration of the variables. Two custom defined objects and a primitive shape (line) are declared. $MenuItem_1$ and $MenuItem_2$ denote the menu items and the line denotes the separator. A menu item is a rectangle containing a text (textrect). Therefore, $MenuItem_1$ is defined as a textrect containing the string 'Parent Directory' and $MenuItem_2$ is defined as a textrect containing the string 'Properties'.

The constraint in $MenuWithSeparator$ states that the menu item 'Properties' should be above the separator which in turn should be above the menu item 'Parent Directory'. The menu item separator is described as a horizontal line. To force this attribute in the constraint, a horizontal line is characterized by the fact that the ordinates y_1 and y_2 are equal and the abscissa x_1 is less than x_2.

After running the application to test the scenario, the instrumented Swing library generates a long output list of shapes being drawn on the screen. The GUICop checker inspects the output of the instrumentation and asserts that the menu item separator is missing, as shown in Figure 11.6.

```
Solving for: ((((m2 above l1) above m1) and ((l1 . y1) == (l1 . y2))) and ((l1 .
     x1) < (l1 . x2)))
Number of logic operations: 13 x 103
Number of endNode components: 0 and Number of endNode strings: 0
-----------------------------------------
object1 is not satisfied
-----------------------------------------
```

Figure 11.6: Output of the checker. Case Jedit Missing Separator.

11.1.3 Missing Hot Key Indicator

We removed a hot key indicator from the Markers menu in jEdit. We then wrote a specification to check for the existence of the hot key indicator and the checker asserts its absence. Figure 11.7(left) shows the Markers menu with the hot key indicator under the letter M.

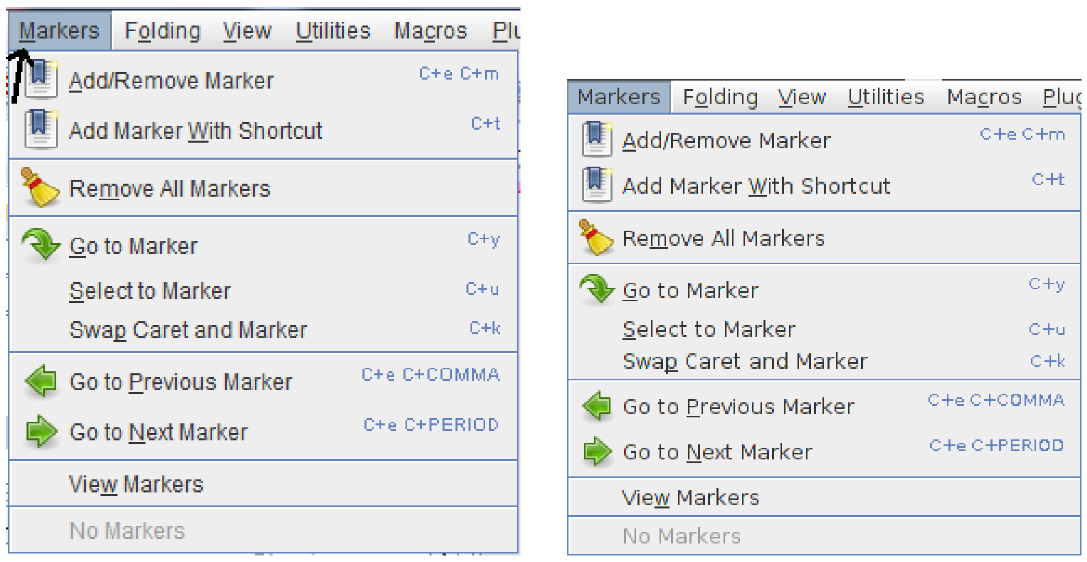

Markers Folding View Utilities Macros Plu
Add/Remove Marker C+e C+m
Add Marker With Shortcut C+t
Remove All Markers
Go to Marker C+y
Select to Marker C+u
Swap Caret and Marker C+k
Go to Previous Marker C+e C+COMMA
Go to Next Marker C+e C+PERIOD
View Markers
No Markers
Markers Folding View Utilities Macros Plug
Add/Remove Marker C+e C+m
Add Marker With Shortcut C+t
Remove All Markers
Go to Marker C+y
Select to Marker C+u
Swap Caret and Marker C+k
Go to Previous Marker C+e C+COMMA
Go to Next Marker C+e C+PERIOD
View Markers
No Markers

Figure 11.7: Markers Menu (left) with a hot key indicator; (right) without the hot key indicator

In the properties file of JEdit, `jedit_en.props`, a hot key is assigned by adding a dollar sign $ next to the letter M in the Markers Menu entry, as per the following:

```
#{{{ Markers menu
markers.label=$Markers
```

After removing the dollar sign $ and rebuilding JEdit with the modified properties file, the Markers Menu no longer has the hot key indicator. Figure 11.7(right) shows the Markers menu without the hot key indicator.

We developed a specification to check whether the hot key indicator under letter M in the Markers menu exists, as shown in Figure 11.8. The specification has two `specobjects`. *HotKeyIndicator* is the `specobject` to be checked.

```
HotKeyIndicator = {
        variables {
                markers m1;
                rectangle hotkey;
        }
        properties {
                X = m1.x;
                Y = m1.y;
                WIDTH = m1.width;
                HEIGHT = m1.height;
        }
        constraints {
                (((((m1 above hotkey) and (hotkey.height < 3)) and (m1.x < (
                    hotkey.x + 1))) and (hotkey.x < (m1.x + 1))) and (hotkey.y
                    == ((m1.y + m1.height) + 1)));
        }
}
markers = {
        variables {
                textrect t1;
        }
        properties {
                X = t1.x;
                Y = t1.y+1;
                WIDTH = t1.width;
                HEIGHT = t1.height-1;
        }
        constraints {
                (t1.string == 'Markers');
        }
}
```

Figure 11.8: Missing Hot Key Indicator Specification.

The specification has a hierarchical composition where *HotKeyIndicator* is defined in terms of the object *markers*.

The first section of the specification of *HotKeyIndicator* consists of the declaration of the variables. A custom defined object and a primitive rectangle are declared. *markers* denotes the Markers menu and the rectangle denotes the hot key or the underscore like shape under the letter M in 'Markers'. The Markers menu is a rectangle containing a text which corresponds to the primitive shape textrect. Therefore, m_1 is defined as a textrect containing the string 'Markers'.

The constraint in *HotKeyIndicator* states that the textrect 'Markers' should be above the underscore like shape of the hot key. The height of the underscore like shape should be very small so that the rectangle looks more like a line. The underscore like shape should start around the beginning of the textrect 'Markers' and should be positioned right below the textrect.

After running the application to test the scenario, the instrumented Swing library generates a long output list of shapes being drawn on the screen. The GUICop checker inspects the output of the instrumentation and asserts that the hot key indicator is missing, as shown in Figure 11.9.

```
Solving for: (((((m1 above hotkey) and ((hotkey . height) < 3)) and ((m1 . x) <
    ((hotkey . x) + 1))) and ((hotkey . x) < ((m1 . x) + 1))) and ((hotkey . y)
    == (((m1 . y) + (m1 . height)) + 1)))
Number of logic operations: 0 x 9
Number of endNode components: 0 and Number of endNode
----------------------------------------
object1 is not satisfied
----------------------------------------
```

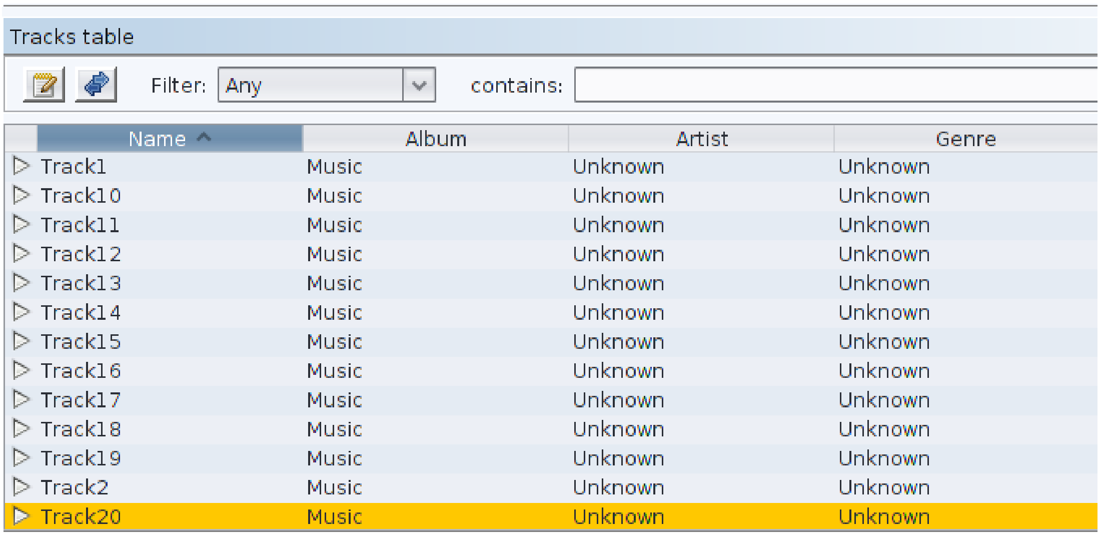
Figure 11.9: Output of the checker. Case Jedit Missing Hot Key Indicator.

11.2 Case Study Advanced JukeBox

Jajuk is a cross-platform Java music organizer and player. It is directed towards advanced users who are looking for powerful functionalities [37]. We wrote a specification to check against an existing Jajuk defect. The defect relates to numbered tracks not being listed in order. We wove the GUICop specification in the corresponding code position in Jajuk. We present below the scenario where GUICop detected the issue.

11.2.1 Unordered Listing of Numbered Tracks

When listing numbered tracks, Jajuk fails to put them in the right order. For instance, 'Track1' is followed by 'Track10' instead of 'Track2'. We reported this issue on GitHub under Issue number 1991 [38]. Figure 11.10 shows how Jajuk lists numbered tracks in the library or the Tracks table. GUICop can detect this issue by asserting that 'Track2' does not precede 'Track10', as shown in the figure.


Tracks table

Filter: Any contains:

Name	Album	Artist	Genre
Track1	Music	Unknown	Unknown
Track10	Music	Unknown	Unknown
Track11	Music	Unknown	Unknown
Track12	Music	Unknown	Unknown
Track13	Music	Unknown	Unknown
Track14	Music	Unknown	Unknown
Track15	Music	Unknown	Unknown
Track16	Music	Unknown	Unknown
Track17	Music	Unknown	Unknown
Track18	Music	Unknown	Unknown
Track19	Music	Unknown	Unknown
Track2	Music	Unknown	Unknown
Track20	Music	Unknown	Unknown

Figure 11.10: Numbered tracks listed in Jajuk.

We developed a specification to check whether the tracks are listed in order, by checking for instance if 'Track2' precedes 'Track10', as shown in Figure 11.11. The specification has three `specobjects`. *NumberedTracks* is the `specobject` to be checked.

```
NumberedTracks = {
        variables {
                secondtrack t1;
                tenthtrack t2;
        }
        properties {
                X = t1.x;
                Y = t1.y;
                WIDTH = t1.width;
                HEIGHT = t1.height+t2.height;
        }
        constraints {
                (t1 above t2);
        }
}
tenthtrack = {
        variables {
                textrect t2;
        }
        properties {
                X = t2.x;
                Y = t2.y;
                WIDTH = t2.width;
                HEIGHT = t2.height;
        }
        constraints {
                (t2.string == 'Track10');
        }
}
secondtrack = {
        variables {
                textrect t1;
        }
        properties {
                X = t1.x;
                Y = t1.y;
                WIDTH = t1.width;
                HEIGHT = t1.height;
        }
        constraints {
                (t1.string == 'Track2');
        }
}
```

Figure 11.11: Numbered Tracks Listing Specification.

The specification has a hierarchical composition where *NumberedTracks* is defined in terms of the objects *secondtrack* and *tenthtrack*.

The first section of the specification of *NumberedTracks* consists of the declaration of the variables. Two custom defined objects are declared. *secondtrack* and *tenthtrack* denote the second and the tenth tracks in the library. The second and tenth tracks are basically textrects, i.e. rectangles containing text, 'Track2' and 'Track10' respectively.

The constraint in *NumberedTracks* states that the second track should be above the tenth track. After running the application to test the secnario, the instrumented Swing library generates a long output list of shapes being drawn on the screen. The GUICop checker inspects the output of the instrumentation and asserts that 'Track2' does not precede 'Track10' as shown in Figure 11.12.

```
Specification error for solving: (t1 above t2)
Number of endNode components: 0 and Number of endNode strings: 0
------------------------------------------
object1 is not satisfied
------------------------------------------
```

Figure 11.12: Output of the checker. Case Jajuk Unordered Listing of Numbered Tracks.

11.3 Case Study Gason

Gason is an open source plugin developed in Java to use sqlmap from BurpSuite [39] which is an integrated platform for performing security testing of web applications [40]. We wrote a specification to check against an existing Gason defect. The defect relates to the labels not displaying words entirely. We wove the GUICop specification in the corresponding code position in Gason. We present below the scenario where GUICop detected the issue.

11.3.1 Cropped Content in Labels

The source code of Gason is hosted on Google Code. A defect is found regarding the user interface of Gason and is reported under $Issue_3$ [41]. This issue consists of the content of the labels being cropped and not seen completely. The labels are not resizable either. Figure 11.13 shows how the main window of Gason appears with the content of the labels not being displayed completely.

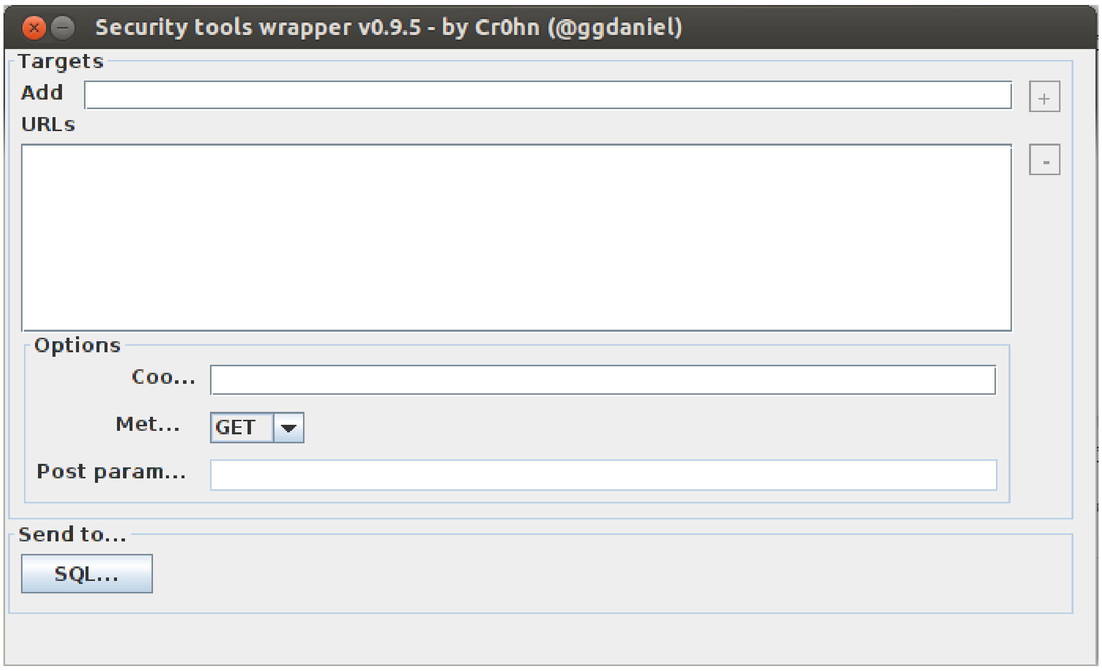


Figure 11.13: Main window of Gason with cropped content in the labels.

We developed a specification to check whether the content of the labels is displayed correctly, by checking for one of the labels containing the word 'Cookie', as shown in Figure 11.14.

```
CroppedContent = {
        variables {
                textrect t1;
        }
        properties {
                X = t1.x;
                Y = t1.y;
                WIDTH = t1.width;
                HEIGHT = t1.height;
        }
        constraints {
                (t1.string == 'Cookie');
        }
}
```

Figure 11.14: Cropped Content Specification

The specification has one `specobject` to be checked, *CroppedContent*. The first section of the specification of *CroppedContent* consists of the declaration of the variables. A textrect is declared to denote a label containing text. The constraint in *CroppedContent* states that the label should contain the word 'Cookie'. After running the application to test the secnario, the instrumented Swing library generates a long output list of shapes being drawn on the screen. Since the word 'Cookie' isn't fully displayed, GUICop detects the issue by looking for the full word in the instrumentation output and reports a failure, as shown in Figure 11.15.

```
Specification error for solving: ((t1 . string) == 'Cookie')
Number of endNode components: 0 and Number of endNode strings: 0
------------------------------------------
object1 is not satisfied
------------------------------------------
```

Figure 11.15: Output of the checker. Case Gason Cropped Content in Labels.

Chapter 12

CONCLUSION

In this work, we presented GUICop an automated GUI testing tool to check whether the execution trace of a GUI program satisfies a user-defined specification. GUICop is a full toolset that handles the testing process from launching the application to driving the application's GUI to finally computing the check without impacting the application's original code. The GUICop specification language is based on basic geometric shapes, events, and relative positional and timing operators. GUICop captures the behavior of the GUI through an instrumented Java Swing library. The GUICop specification language and the supporting solver tolerate non-functional behavioral differences such as screen resolution and color schemes. This allows for more robust testing suites.

GUICop was successfully evaluated using real life case studies and was seen effective at detecting layout problems undetectable in the widgets properties. In the future we will support more shapes and operators with GUICop and provide a more extensive library of GUI objects. We will instrument other graphics libraries. Furthermore, we will extend the functionality of the driver by expanding the list of executable widgets and generating automated test cases to exercise the GUI.

Appendix A

JEdit

A.1 Building jEdit

A build.xml file that comes with the source code of jEdit enables the user to compile the code and run the tool through ANT. A few changes were performed to the build.xml as shown in Figure A.1:

- Addition of the guicop.jar and antlr-3.4.jar to the classpath of the javac and java commands in the "compile" and "run" targets.
- Addition of the JVM argument to the java command in the "run" target in order to load the instrumented graphics library

```
<target name="compile" depends="init">
                <mkdir dir="${classes.dir}/core"/>
                <depend srcDir="${basedir}"
                        destDir="${classes.dir}/core"
                        cache="${classes.dir}"/>
                <dependset>
                        <srcfilelist files="build.xml"/>
                        <targetfileset dir="${classes.dir}/core"/>
                </dependset>
                <javac srcdir="${basedir}"
                       destdir="${classes.dir}/core"
                       debug="true"
                       debuglevel="${config.build.debuglevel}"
                       nowarn="${config.build.nowarn}"
                       deprecation="${config.build.deprecation}"
                       source="${target.java.version}"
                       target="${target.java.version}"
                       compiler="modern"
                       encoding="UTF-8"
                       includeAntRuntime="false">
                        <classpath id="classpath.compile">
                                <fileset dir="${lib.dir}/compile"
                                         includes="*.jar"/>
            <!-- Needed jar files added here -->
                                <fileset dir="/home/dalal/workspace/Examples/bin
                                    "
                                         includes="*.jar"/>
                        </classpath>
                        <include name="org/**"/>
                        <compilerarg line="${config.build.compilerarg}"/>
                </javac>
        </target>

<target name="run"
                depends="init,build"
                description="run jEdit">
 <java jar="${jar.location}/${jar.filename}"
                      fork="true"
                      spawn="true"
            <!-- Needed jar files added here-->
                        classpathref="classpath.compile">
            <!-- Loading the instrumented graphics library -->
                        <jvmarg value="-Xbootclasspath/p:/home/dalal/Desktop/
                            Research/rt.jar"/>
                        <arg value="-settings=${build.dir}/settings"/>
                </java>
</target>
```

Figure A.1: Modifications made to the build.xml file of jEdit

A.2 Calling GUICop

Upon closing the about dialog in the help menu of jEdit, a call to the GUICop checker is woven to check for the provided specification against the instrumentation output. The call for GUICop is made within the `closeDialog()` method of the

`AboutDialog.java` class, before the dialogue is completely disposed. The code snippet is shown in Figure A.2. The code is applied across all case studies to call the checker.

```
        {
                try {
                Thread.sleep(2000);
            } catch (InterruptedException ex) {
//                 Logger.getLogger(ToolBarDemo.class.getName()).log
        (Level.SEVERE, null, ex);
            }

            try {
                 Class c = guicop.Main.class;
                 Class[] argTypes = new Class[]{String[].class};
                 Method main;
                 obj = c.newInstance();
                 main = c.getDeclaredMethod("main", argTypes);
                 System.out.format("invoking %s.main()%n", c.getName
                     ());
                 main.invoke(obj, new String[1]);
                } catch (NoSuchMethodException | SecurityException
                    e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                } catch (IllegalAccessException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                } catch (IllegalArgumentException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                } catch (InvocationTargetException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                } catch (InstantiationException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                }
        }
```

Figure A.2: Java code to call the GUICop checker

Bibliography

[1] D. Mounty and M. O'Connor, "Automating execution of arbitrary graphical interface applications," Jul. 29 2014, US Patent 8,793,578. [Online]. Available: http://www.google.com/patents/US8793578

[2] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using GUI Screenshots for Search and Automation," in *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '09. New York, NY, USA: ACM, 2009, pp. 183–192. [Online]. Available: http://doi.acm.org/10.1145/1622176.1622213

[3] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI Testing Using Computer Vision," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 1535–1544. [Online]. Available: http://doi.acm.org/10.1145/1753326.1753555

[4] U. Kanewala and J. M. Bieman, "Techniques for testing scientific programs without an Oracle," in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering, SE-CSE 2013, San Francisco, California, USA, May 18, 2013*, 2013, pp. 48–57. [Online]. Available: http://dx.doi.org/10.1109/SECSE.2013.6615099

[5] A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–. [Online]. Available: http://dl.acm.org/citation.cfm?id=950792.951350

[6] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jRapture: A Capture/Replay Tool for Observation-based Testing," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '00. New York, NY, USA: ACM, 2000, pp. 158–167. [Online]. Available: http://doi.acm.org/10.1145/347324.348993

[7] A. M. Memon and M. L. Soffa, "Regression Testing of GUIs," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-11. New York, NY, USA: ACM, 2003, pp. 118–127. [Online]. Available: http://doi.acm.org/10.1145/940071.940088

[8] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Plan Generation for GUI Testing," in *Proceedings of The Fifth International Conference on Artificial Intelligence Planning and Scheduling.* AAAI Press, Apr. 2000, pp. 226–235.

[9] ——, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 144–155, Feb. 2001. [Online]. Available: http://dx.doi.org/10.1109/32.908959

[10] Q. Xie and A. M. Memon, "Designing and Comparing Automated Test Oracles for GUI-based Software Applications," *ACM Trans.*

Softw. Eng. Methodol., vol. 16, no. 1, Feb. 2007. [Online]. Available: http://doi.acm.org/10.1145/1189748.1189752

[11] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, “GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software,” *Automated Software Engg.*, vol. 21, no. 1, pp. 65–105, Mar. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10515-013-0128-9

[12] A. M. Memon, M. L. Soffa, and M. E. Pollack, “Coverage Criteria for GUI Testing,” in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-9. New York, NY, USA: ACM, 2001, pp. 256–267. [Online]. Available: http://doi.acm.org/10.1145/503209.503244

[13] “Swing GUI Toolkit Group.” [Online]. Available: http://openjdk.java.net/groups/swing/

[14] “Java AWT Reference.” [Online]. Available: http://www.oreilly.com/openbook/javawt/book/

[15] “A Swing Architecture Overview.” [Online]. Available: http://www.oracle.com/technetwork/java/architecture-142923.html

[16] R. Eckstein, M. Loy, and D. Wood, *Java Swing.* Sebastopol, CA, USA: O’Reilly & Associates, Inc., 1998.

[17] “Class Graphics2D.” [Online]. Available: http://docs.oracle.com/javase/8/docs/api/java/awt/Graphics2D.html

[18] "Java 2DTM API Overview." [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-intro.html

[19] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 2007.

[20] "Introduction to Aspect-Oriented Programming." [Online]. Available: http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html

[21] A. Ruiz and Y. W. Price, "Test-Driven GUI Development with TestNG and Abbot," *IEEE Softw.*, vol. 24, no. 3, pp. 51–57, May 2007. [Online]. Available: http://dx.doi.org/10.1109/MS.2007.92

[22] A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces," Ph.D. dissertation, 2001, aAI3026063.

[23] "Abbot." [Online]. Available: http://abbot.sourceforge.net/

[24] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment," *Commun. ACM*, vol. 41, no. 5, pp. 81–86, May 1998. [Online]. Available: http://doi.acm.org/10.1145/274946.274960

[25] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing Test Cases for Regression Testing," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '00. New York, NY, USA: ACM, 2000, pp. 102–112. [Online]. Available: http://doi.acm.org/10.1145/347324.348910

[26] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon, "Conceptualization and Evaluation of Component-Based Testing Unified with Visual GUI Testing:

An Empirical Study," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, April 2015, pp. 1–10.

[27] A. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *Software Engineering, IEEE Transactions on*, vol. 31, no. 10, pp. 884–896, Oct 2005.

[28] A. M. Memon, "An Event-flow Model of GUI-based Applications for Testing: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1002/stvr.v17:3

[29] Q. Xie and A. M. Memon, "Rapid "Crash Testing" for Continuously Evolving GUI-Based Software Applications," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 473–482. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2005.72

[30] P. Aho, T. Raty, and N. Menz, "Dynamic reverse engineering of GUI models for testing," in *Control, Decision and Information Technologies (CoDIT), 2013 International Conference on*, May 2013, pp. 441–447.

[31] Y. Miao and X. Yang, "An FSM based GUI test automation model," in *Control Automation Robotics Vision (ICARCV), 2010 11th International Conference on*, Dec 2010, pp. 120–126.

[32] V. Lelli, A. Blouin, and B. Baudry, "Classifying and Qualifying GUI Defects," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, April 2015, pp. 1–10.

[33] "ECLIPSE WIKI." [Online]. Available: https://wiki.eclipse.org/

[34] M. Robinson and P. Vorobiev, *Swing.* Manning Publications, 2003.

[35] "The Java Tutorials." [Online]. Available: https://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html

[36] "jEdit Features." [Online]. Available: http://www.jedit.org/?page=features#modes

[37] "Jajuk Advanced Jukebox." [Online]. Available: http://www.jajuk.info/

[38] "Numbered Tracks not in Order." [Online]. Available: https://github.com/jajuk-team/jajuk/issues/1991

[39] "Sqlmap plugin for BurpSuite." [Online]. Available: http://blog.buguroo.com/sqlmap-plugin-for-burpsuite/?lang=en

[40] "BurpSuite." [Online]. Available: https://portswigger.net/burp/

[41] "Issue 3: [GUI] Contents of Listboxes cannot been seen completely." [Online]. Available: https://code.google.com/p/gason/issues/detail?id=3