

GUITAR: an innovative tool for automated testing of GUI-driven software

Bao N. Nguyen · Bryan Robbins · Ishan Banerjee · Atif Memon

Received: 16 April 2012 / Accepted: 25 April 2013 / Published online: 7 May 2013
© Springer Science+Business Media New York 2013

Abstract Most of today's software applications feature a graphical user interface (GUI) front-end. System testing of these applications requires that test cases, modeled as sequences of GUI events, be generated and executed on the software. We term *GUI testing* as the process of testing a software application through its GUI. Researchers and practitioners agree that one must employ a variety of techniques (e.g., model-based, capture/replay, manually scripted) for effective GUI testing. Yet, the tools available today for GUI testing are limited in the techniques they support. In this paper, we describe an innovative tool called GUITAR that supports a wide variety of GUI testing techniques. The innovation lies in the architecture of GUITAR, which uses plug-ins to support flexibility and extensibility. Software developers and quality assurance engineers may use this architecture to create new toolchains, new workflows based on the toolchains, and plug in a variety of measurement tools to conduct GUI testing. We demonstrate these features of GUITAR via several carefully crafted case studies.

Keywords GUI testing · Test automation · Test generation

B.N. Nguyen (✉) · B. Robbins · I. Banerjee · A. Memon
Department of Computer Science, University of Maryland, College Park, MD 20742, USA
e-mail: baonn@cs.umd.edu

B. Robbins
e-mail: brobbins@cs.umd.edu

I. Banerjee
e-mail: ishan@cs.umd.edu

A. Memon
e-mail: atif@cs.umd.edu

1 Introduction

Graphical User Interfaces (GUIs) provide the primary user interface in the vast majority of today's commodity software (Myers 1995; Memon and Nguyen 2010; Shneiderman et al. 2009; Silva et al. 2008). We term a *GUI-driven software application* (or simply *GUI software* or *GUI application*) as a software application program with a GUI as its main interface for interaction. We consider GUI software to be a subset of event-driven systems (Memon 2007; Ural and Yang 1991), also called reactive systems (Veanes et al. 2008).

In GUI software, interface components form the visible GUI structure, and these components accept sequences of user events (e.g., mouse clicks, type-in-text) that alter the state of the software. This modification of state may or may not include a change to the visible GUI itself. The testing of GUI software, then, involves executing events belonging to GUI components and monitoring resulting changes to the program state. GUI test cases therefore have event sequences as input and some indication of a program's state (e.g., GUI state, memory state, error log, or any other indicator of runtime application state) as expected output (Artzi et al. 2011; Mesbah and van Deursen 2009; Xie and Memon 2007).

Because GUIs exist at the point of the end user interaction, *GUI testing* represents a form of system-level testing for GUI software. GUI testing actually tests much more than the code associated with the GUI itself. Even though test cases execute on the GUI and output may be extracted from the GUI layer, prior work has shown that this type of testing is effective at detecting both GUI and non-GUI faults (Robinson et al. 2008; Brooks et al. 2009). This is because the events of the GUI execute underlying non-GUI code. In many cases—unless an application includes both a GUI and a non-GUI interface—GUI testing is the only form of system-level testing possible for GUI software. This makes GUI testing a critical part of testing for any GUI software.

The size and complexity of modern GUIs, in terms of GUI components, and the events that may be executed on them, exceed the practical limits of exhaustive and analytical approaches to testing (Belli 2001; Memon 2007). The number of possible test cases for a GUI increases exponentially with the number of events per test case. Even a trivial GUI with a single repeatable event has infinitely many test cases, because an end-user may perform that event an arbitrary number of times. Each such sequence is a potential test case. Unfortunately, testers cannot ignore long and complex event sequences, because these sequences often reveal state- and context-specific bugs that require chaining multiple GUI events together (Xie and Memon 2008; Yuan and Memon 2010).

In practice, GUI testing tools use two major approaches. The first and most popular approach is to employ a *script-based* language such as JFCUnit,¹ Selenium Web-

¹<http://www.jfcunit.sourceforge.net>.

Driver,² Robotium,³ Abbot,⁴ or SOAtest⁵ to manually create unit test cases for GUIs. A unit test case consists of method calls which programmatically invoke GUI events. The tests record and verify output using tester-specified assertions. Because manual coding of test cases can be tedious, an alternative approach called *capture/replay* is also used. Examples of capture/replay tools include Test Automation FX,⁶ Selenium IDE,⁷ and Quick Test Pro.⁸ The test tool first “captures” sequences of events that testers perform *manually* on the GUI. These sequences are treated as test cases for testing the GUI. The test cases can be later “replayed” automatically on the GUI to detect bugs. Both *script-based* and *capture/replay* tools offer limited test automation. These tools require manual test case creation, which is expensive and typically leads to a small number of test cases.

In this paper, we posit that the complex nature and pervasive use of modern GUIs requires flexible tool support for testing. A flexible GUI testing tool would allow the tester to tailor the tool’s capabilities for the needs of specific applications. The tester, with knowledge of the GUI’s intricacies, would be in a position to customize the tool’s algorithms while having the option of using the tool as a part of a larger test harness.

We describe GUITAR, an innovative and flexible framework for the development of GUI testing tools. By *framework* we mean that GUITAR supports the development of customized tools for testing. Over the years we, and others affiliated with our research group, have developed many open-source tools based on the GUITAR framework. We openly distribute many of these tools at <http://guitar.sourceforge.net>. The design and implementation of GUITAR emphasizes the following features:

- *Automation*: GUITAR automates key GUI testing activities while lending itself well to third-party test harnesses.
- *Algorithm Reuse*: GUITAR’s core is not limited to a specific application, platform, or technique, but maximizes reuse of platform-independent aspects of algorithms.
- *Algorithm Customization*: GUITAR supports the customization of tools for specific quality assurance goals (e.g., on specific platforms, with specific applications, with specific types of monitoring and oracles, etc.).
- *Model-based*: GUITAR is natively but not exclusively model-based.
- *Modularity*: GUITAR’s architecture is plugin-based.

In Sect. 2, we discuss GUI testing activities and related literature. In Sect. 3, we discuss core algorithms and examine GUITAR’s standard use case with a case study of GUITAR’s Java Foundation Classes (JFC) based toolchain. We then review the GUITAR architecture (Sect. 4), which sets up our discussion of five additional

²<http://www.seleniumhq.org/projects/webdriver>.

³<http://www.code.google.com/p/robotium>.

⁴<http://www.abbot.sourceforge.net>.

⁵<http://www.parasoft.com/jsp/products/soatest.jsp>.

⁶<http://www.testautomationfx.com>.

⁷<http://www.seleniumhq.org/projects/ide>.

⁸<http://www.hp.com/QuickTestPro>.

case studies with GUITAR (Sect. 5). Finally, we conclude by evaluating GUITAR's strengths and weaknesses, and discuss additional uses from literature and future work (Sect. 6).

2 Background and related work

Before we discuss the design and use of GUITAR, we first consider the activities associated with software testing in general, and GUI testing in particular. We also discuss related, existing work on GUI testing.

2.1 GUI testing activities

At a minimum, testers must *construct test cases*, combine test cases to *construct test suites*, and then *execute test cases* on applications to obtain test results. We consider each of these activities in the specific context of GUI testing, though many aspects apply to testing in general.

2.1.1 Test case construction

As mentioned earlier, test case construction for GUI testing involves selecting sequences of GUI events and describing the expected state of the program after the events' execution. We refer to these two activities as *event selection* and *oracle specification*, respectively.

Event selection involves composing sequences of GUI events that make useful test cases. Often, test cases follow an existing system-level specification, such as a set of detailed use cases or performance requirements (Silva et al. 2008; Memon et al. 2001a). Just as in every level of testing, GUI tests must consider both valid and invalid inputs. From a tool perspective, many tools exist for capturing manually entered sequences. We discuss these alternatives in Sect. 2.2. In this “capture/replay” paradigm, test case selection involves entering every input sequence of events manually.

While the capture/replay approach offers complete control over every aspect of the selection process and sets up automated test case execution, we argue that the size of any non-trivial GUI often makes completely manual capturing intractable. For example, the version of Microsoft WordPad in Windows 7 (considered to be a rather simple GUI by today's standards) contains over 50 GUI events. This number may seem small, but the number of possible test cases increases exponentially when considering combinations of these 50 events. In these cases, testers need a more complete consideration of the GUI's event space. Model-based testing (MBT) approaches for GUI-driven software construct an abstraction of some subset of an application and prescribe a test case selection process which constructs test cases based on the model (Silva et al. 2008; Mesbah and van Deursen 2009; Jääskeläinen et al. 2009; Memon 2007; Belli 2001). MBT approaches may or may not be more tractable than capture-based alternatives, depending on the cost of constructing the model and selecting test cases. We argue that these approaches offer more opportunities for automation than a completely manual selection process.

Oracle specification involves the specification of expected output for a test (Baresi and Young 2001; Xie and Memon 2007). Tests may specify expectations at many possible levels and at many possible points during a GUI test. A simple oracle may perform *crash testing*, in which the test passes if no errors occur during the program's execution. Other possible assertions include the specification of a dynamic aspect of the program's state, such as a stack trace. In GUI testing, one may also assert the state of the GUI, e.g., "*The JTextArea should contain no text after the test case executes.*"

Test oracle precision, while sometimes necessary and certainly capable of finding bugs, have drawbacks. First, the use of more specific oracles limits opportunities for automation. The automation of crash testing only involves examining a log, but expectations of GUI and application state upon test completion require an existing "gold standard". As developers introduce intentional changes to a program, testers must constantly update test oracles, leading to a higher cost of test maintenance.

Capture/replay tools as well as MBT approaches (discussed so far) do not avoid these oracle specification difficulties (Xie and Memon 2007; Mesbah and van Deursen 2009). At best, tools can assist in the creation of oracles based on the runtime state of an application (e.g., by saving program state as a reference). Such an oracle may be used for regression testing future versions of the application. This type of oracle can only automatically determine whether the application's output has changed or not, which may or may not indicate a functional error.

2.1.2 Test suite construction

Test suite construction involves selecting a set of test cases, presumably from a pool of existing or possible test cases, to provide the desired level of quality assurance for an application. While test suite requirements follow from specifics about an application and its quality assurance goals, test suite construction in general centers around a common decision point: *How much testing is enough?*

Testers commonly build suites around the notion of *coverage*, in order to guarantee that a test suite covers a certain percentage of source code artifacts (methods, lines, branches, or classes) of a program (Zhang et al. 2011; Cadar et al. 2008). For GUI testing, previous work by our group has introduced the notion of event coverage, which considers a suite's coverage of certain types of events, event combinations, such as all pairs of events (Memon et al. 2001b), or events in different contexts (Yuan et al. 2011). The execution of all possible test cases of a significant length is rarely possible in real GUI-driven software. Therefore, the tradeoff between adequacy measures such as coverage and the availability and cost effectiveness of resources plays a large role in GUI test suite construction.

MBT approaches allow for a systematic approach to test suite construction (e.g., event coverage derived from an event-based model). In a capture-based or completely manual approach, coverage may be considered after the fact using off-the-shelf tools, but without a model of events or application states, coverage is likely limited to code-based metrics (Ganov et al. 2008; Staiger 2007).

2.1.3 Test execution

Test execution involves the execution of previously defined test cases. Most testing efforts utilize scripted test suites, as supported by a capture/replay or MBT tool. With

GUI testing, the primary difficulty in automatic replay arises from widget (or GUI component) identification (Ruiz and Price 2008; McMaster and Memon 2009). In GUI testing, any widgets with an associated event should be eligible for automatic interaction with a replay tool. The most basic GUI test execution tools use coordinate information as a basis for performing user events, but coordinate information ties replay tools to a specific architecture and pixel resolution. The GUI must render exactly the same for each test case execution so that widgets are identified correctly. More general tools use additional GUI properties for identification, such as a widget's parent component or window, its color, and label.

2.2 Related work

In our discussion of testing activities, we mentioned current approaches to GUI testing, such as MBT and capture/replay. In this section, we provide more details of existing tools and approaches relevant to our research. Several GUI testing frameworks and associated toolchains have been developed for industrial and academic use. Based on the underlying test case generation techniques, these frameworks may be grouped into four broad categories: *script-based*, *capture/replay*, *random-walk*, and *automated model-based*.

Script-based frameworks provide scripting languages to programmatically control the GUI. Most languages extend or adapt the JUnit framework,⁹ developed for unit testing. Using the provided languages, testers can write test scripts to automatically interact with the GUI. Test cases assert whether the application executed correctly. During execution, assertion violations report any errors as test case failures. Script-based tools are widely used in the industry. Examples of such tools include JFCUnit, Selenium WebDriver, Robotium, Abbot, and SOAtest.

Academic researchers have also developed similar tools for research purposes. For example, GTT (Chen et al. 2005) supports test-driven development of Java-based GUI applications. It provides visual editing of test scripts, enabling test-driven development in extreme programming environments. As another example, the Sikuli testing framework (Chang et al. 2010) employs computer vision techniques to develop a visual language for writing test scripts. The testers can use images of widgets to visually identify GUI components and drive test case execution instead of using their textual labels or other properties.

Using scripted test cases can significantly reduce GUI testing effort. Once test cases are developed, they may be reused to automate GUI interactions in regression testing. However, because script writing is a very labor-intensive process, the number of test cases is often small. *Capture/replay* tools can reduce the burden of scripting by providing interactive tool support. Examples of current capture/replay tools include Test Automation FX, Selenium IDE, Rational Robot,¹⁰ HP WinRunner¹¹ and Quick Test Pro. The tools provide mechanisms for testers to record interactions with the GUI and save their interaction as test cases. These test cases may be replayed automatically

⁹<http://www.junit.org>.

¹⁰<http://www.ibm.com/software/awdtools/tester/robot>.

¹¹<http://www.hp.com/functionaltesting>.

to perform testing. However, because the capture phase is still done manually, these tools typically yield a small number of test cases.

Random-walk tools such as Android Monkey¹² and GUIDancer¹³ are used for crash testing. Unlike script-based and capture/replay tools, random-walk tools do not generate test cases. Instead, they randomly walk the user interface and execute all encountered events in sequence. These tools are easy to use and may indeed find bugs by using unexpected combinations of events, but with no notion of a test case or the ability to replay exact sequences, they typically supplement a broader approach to testing.

Recently, there have been some advances in developing *automated model-based* testing tools. These tools create a model of the GUI and use this model to automatically generate test cases. Tools differ in terms of the underlying models used and the processes used to obtain, manipulate, and extract test cases and other information from the models. TEMA (Jääskeläinen et al. 2009) is a MBT framework developed for smartphone applications. The tester *manually* creates a two-tier model consisting of two state machines called *action* and *keyword* machines, which represent the GUI at design and implementation levels, respectively. The method traverses the action machine to generate design-level test cases, then uses the keyword machine to transform design test cases into executable ones.

MBT approaches employ tool support for automated test case generation. Several tools exist for generating test cases automatically. PATHS (Memon et al. 2001a) uses AI planning to generate test cases based on the state of a GUI before and after executing a user-defined operation. GTG (Nguyen et al. 2010) enables testers to convert business logic test cases into presentation logic test cases. Testers specify a mapping from business logic to presentation logic which supports the conversion. PETTool (Cunha et al. 2010) identifies patterns in GUIs and generates generic testing solutions based on the patterns. The symbolic execution tool Barad (Ganov et al. 2008) solves constraints between text input data with the aim of producing a minimal test suite which maximizes code coverage.

MBT approaches can also use tools to support the construction of their model. Several tools exist which support extracting information from the GUI of an application. This information is used for modeling and generating test cases for the application. GUISurfer (Silva et al. 2009) automatically reverse engineers a behavioral model of the GUI from the source code of Java Swing-based GUI applications to produce a Haskell specification. This model of the application is then validated by running test cases. REGUI2FM (Paiva et al. 2008) reverse engineers a GUI into a Spec# model, which can be used in Spec Explorer (Veanes et al. 2008) to generate test cases. It also captures user actions into scenarios which are used for testing.

A^2T^2 (Amalfitano et al. 2011) is a similar MBT tool, but uses a reverse engineering technique to automatically construct the GUI model. This tool leverages a crawl-based technique to reverse engineer the GUI structure of the Android phone application and create a state machine model. The tool automatically generates test cases from the state machine whose results can be automatically checked against

¹²<http://developer.android.com/guide/developing/tools/monkey.html>.

¹³http://www.bredex.de/web/index.php/guidancer_jubula_en.html.

pre-defined constraints. Tools such as Crawljax (Mesbah and van Deursen 2009) and Revangie (Draheim et al. 2005) employ similar techniques for web applications.

Model-based approaches can also use tool support to edit models directly. Certain tools allow the tester to *visually* inspect and edit GUI specifications. The NModel framework¹⁴ provides tools for scripting and visualization of scripted models of C# programs as well as test case generation and replay tools. VESP (Chen and Subramaniam 2001) allows a tester to visually manipulate test specifications of Java-based GUI applications represented as state machines. VESP executes the applications under its control and allows the tester to edit the state machine model directly. The modified state machine model can then be used for test case generation. *ActiveStory Enhanced* (Hellmann et al. 2010) supports agile GUI design and development. Developers create low fidelity prototypes of GUI components which can be used for usability analysis. The low-fidelity design can be used to generate test cases for iterative agile development.

Finally, tools also exist for test maintenance based on model evolution. TDE/UML (Vieira et al. 2006) acts as a plugin to modeling environments such as Rational Rose and provides support for test case generation as part of a comprehensive approach to application modeling. REST (Grechanik et al. 2009) enables a user to evolve test scripts when the GUI changes. The tool detects differences between original and modified versions of a GUI and generates a warning if a script needs correction.

2.3 Comparing GUITAR to existing frameworks

To provide appropriate context for our discussion of GUITAR, we now consider how the approach of GUITAR, from an automation engineering perspective, compares to existing alternatives. Here we focus on four alternative frameworks that, to our knowledge, are individually among the most actively used in practice and as a group provide a representative sample across the different approaches to testing we previously reviewed.

1. Monkey, a tool for random walking of Android application GUIs
2. NModel, a model-based testing framework for C# programs
3. Quick Test Pro, a popular proprietary, multi-platform tool for test automation
4. Selenium WebDriver, a popular API for browser automation.

Table 1 offers a concise, high-level comparison of the frameworks. While a full experimental comparison of these tools is outside the scope of the current paper, comparison of the features of the frameworks highlights some unique features of GUITAR.

Monkey, as a random walking tool for Android mobile applications, uses no underlying model. Test cases are not explicitly generated, but instead, the tool executes random events automatically in a single application instance. This provides a simple, fully automated testing tool. For simple mobile applications, this may provide a useful amount of coverage, but without modeling or test case output other than a log of interactions. Monkey compares most directly to the GUITAR Ripper, though

¹⁴<http://nmodel.codeplex.com/>.

Table 1 Comparison of testing frameworks

Framework name	Model generation	Model verification	Test case generation	Test oracles	Supported platforms
GUITAR	Rev. Eng (A)	Manual	Model based (A)	Custom	Multiple [†]
Monkey	None	None	None	Supported events	Android
NModel	Scripted (M)	Tool support (M)	Model based (A) & Scripted (M)	Custom	C#
Quick Test Pro	None	None	Scripted (M) & Captured (M)	Custom	Multiple [‡]
Selenium	None	None	Scripted (M) & Captured (M)	Custom	Web

A = automated, M = manual

[†] JFC, SWT, Web, Android, other platforms in alpha

[‡] Java, Web, .NET

the Ripper algorithm proceeds in a consistent order rather than randomly. Monkey is also the only framework that does not support custom test oracles, as only specific tool-supported error events can be detected (e.g., crashes, timeouts, permissions errors). We currently distribute an Android GUITAR toolchain, which, if combined with our random test case generator working from a model, can also test an Android application by random walk.

The NModel framework provides an interesting comparison for GUITAR. The multiple NModel tools work from a manually provided model specification. Visualization tools assist with model verification, and test case generation and execution follow from the model as defined. NModel only works for C# programs. The tools of NModel compare almost directly to GUITAR's tools except for the absence of a Ripper tool. The scripting alternative allows for more precise model definition at the expense of scalability, as manual specifying a model correctly may take a great deal of time for non-trivial applications. With NModel, assertions can be coded into the model itself and applied during test execution as a test oracle.

Interestingly, the NModel framework contains tool support for model verification. While GUITAR as a framework does not natively support visualization (which allows manual verification of models), the XML formats we use in GUITAR's input and output files support direct conversion into visualizations. We use external processing scripts to generate Graphviz¹⁵ and Gephi¹⁶ visualizations of our models (such as the images included in this paper). We recognize that tool support is often still necessary beyond simply visualizing a model, as the models can be quite large.

The proprietary Quick Test Pro tools support both scripting and capturing of test cases without consideration of a model. The tools can replay existing test cases. As a proprietary tool, developers have less lower-level control and limited ability for

¹⁵<http://www.graphviz.org/>.

¹⁶<https://gephi.org>.

customization. However, the tool does exist for many platforms and continues to expand.

Finally, the Selenium framework provides multiple tools for browser-based testing of web applications. The Selenium IDE provides a plugin for the Firefox browser that supports capture of events. Alternatively, Selenium Web Driver provides programming interfaces in many languages for scripting tests. The scripted tests can then be replayed as an executable program in the chosen language. Selenium also supports a “Grid” mode to run many browser instances and improve scalability. No known Selenium tools work from a model of the application. With Selenium, the application programming interface (API) exposes many types of assertions that can be applied during test execution, coupling the test oracle with the sequence of events. We currently distribute a web version of GUITAR which leverages Selenium’s Firefox Web Driver as a GUI automation backend.

By comparison, GUITAR is uniquely both model-based and multi-platform, as supported by its plugin-based architecture. The plugin-based approach also allows GUITAR to be extended to new platforms as they become available, and allows developers to customize and extend GUITAR at the model, event, and widget levels of abstraction, as we illustrate throughout this paper.

GUITAR supports test oracles capable of verifying any detectable aspect of GUI state both during and after test execution. For runtime assertions, GUITAR requires the implementation of Test Monitor plugins. We describe the implementation of a Test Monitor plugin for collecting code coverage in Case Study 3. GUITAR’s approach to test oracles is similar to the Monkey approach, in that failures of a certain class (e.g., crashes, permissions errors, timeouts) can be detected by a class-specific Test Monitor. However, Test Monitor implementations, because they are plugins to the framework, are completely customizable.

GUITAR’s support for customization throughout the model-based testing process also allows the framework to support development of tools with capabilities very similar to others considered here. We claim that GUITAR’s combination of model-based, multi-platform support, and customization throughout the testing process makes GUITAR an innovative tool as compared to current approaches in the industry.

One weakness of GUITAR evident in this comparison is a lack of tool-based support for any manual (scripted or captured) test development, including integration of manual and model-based test cases. We recognize this as a current issue in GUITAR, and are in the early stages of bringing test case capture capability to the framework in a way that preserves the plugin-based architecture.

3 Using GUITAR: an in-depth case study

We now present GUITAR, our framework for automated GUI testing. The salient features of GUITAR include (1) *GUI reverse engineering*, (2) *automated test case generation*, (3) *automated execution* of test cases, (4) support for *platform-specific customization*, (5) support for addition of new algorithms as *plugins*, and (6) support for *integration* into other test harnesses and quality assurance workflows. To the

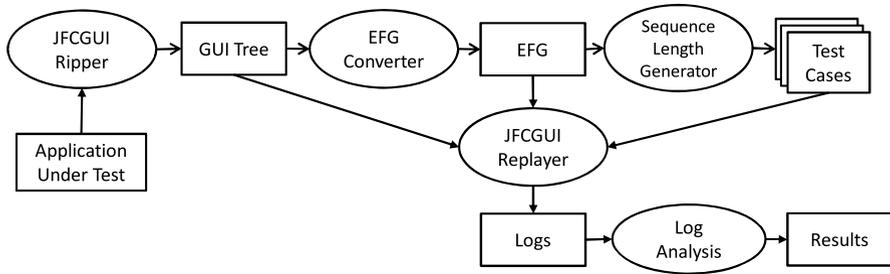


Fig. 1 Standard workflow using JFC toolchain

best of our knowledge, GUITAR is the first GUI testing tool to combine these six capabilities.

For ease of presentation, we demonstrate GUITAR using a workflow called *Case Study 0*. More specifically, we describe a part of GUITAR that is tailored for Java Foundation Classes (JFC), a graphical framework for building portable Java GUIs. We create a simple regression-testing workflow referred to as the *standard workflow* in the rest of this paper. The standard workflow is shown in Fig. 1. We use two open-source JFC-based applications for this demonstration—ArgoUML,¹⁷ which provides graphical tools for working with the Unified Modeling Language (UML), and JabRef,¹⁸ a text-intensive application for citation management. Both ArgoUML and JabRef are mature and popular applications, both around for approximately 10 years and each downloaded more than a million times.

3.1 Components and algorithms

The GUITAR framework allows for the development of four primary tools. They are:

Ripper, a tool for generating a structural model of the GUI of an application under test (AUT) by means of reverse engineering from the run-time state of the application (Memon et al. 2003).

Graph Converter, a tool for converting the structural model, generated by the *Ripper*, into a graph, such as the Event-Flow Graph (EFG).

Test Case Generator, a tool for automated test case generation based on the graph generated by the *Graph Converter*.

Replayer, a tool for automated execution of test cases generated by the *Test Case Generator* on the AUT.

The *Ripper* and the *Replayer* interact with the AUT and hence always require platform-specific customization. The *Graph Converter* and the *Test Case Generator*, on the other hand, process models and are hence platform-independent.

¹⁷<http://argouml.tigris.org>.

¹⁸<http://jabref.sourceforge.net>.

3.1.1 Ripper

The primary purpose of the *Ripper* is to *discover* as much structural information about the GUI as possible using automated algorithms and some human input. This information is output into a structure called a GUI Tree.

During ripping, the GUI application is executed automatically; the application's windows are opened in a depth-first manner. The *Ripper* extracts all the widgets and their properties from the GUI. Properties of widgets include basic attributes such as position, color, size, and enabled status. Properties also include information about widgets' events, such as: whether a widget opens a modal or modeless¹⁹ window or a menu, whether a widget closes a window, and whether the widget is a button or an editable text-field. The *Ripper* extracts properties for widgets as well as their containing GUI windows and stores the information in the GUI Tree. For each GUI window, the *Ripper* first extracts structural information of that window. It then executes widgets that invoke other GUI windows. The depth-first traversal terminates when all GUI windows are covered. Note that the order of invoking the widgets may affect the GUI structure being extracted. Each possible order may result in a slightly different structure. A hypothetical ideal *Ripper* would potentially require infinite sequences of invocations because there are, in principle, an infinite number of ways to interact with a GUI application. This would result in an intractable ripping algorithm for any non-trivial GUI. It is possible to improve accuracy of the *Ripper* by attempting different orders of invocation. Different heuristics will be developed in future work.

Because of the low-level nature of the *Ripper*'s algorithms (e.g., extracting GUI windows and widgets at run-time), it is necessarily a platform-specific component of GUITAR. A plugin customized for a particular platform uses the *Ripper* framework to interact directly with the application on a specific platform. For example, the JFCGUIRipper (discussed later in Sect. 3.2.1) is only able to interact with JFC-based widgets and windows.

Several pieces of information required during ripping must be provided as human input. First, human input directs the unambiguous identification of windows using their titles. Most GUI windows in an application have fixed titles (e.g., *FileOpen* in MS Word, *Print* in Adobe Acrobat). Once the *Ripper* encounters such fixed-title windows, it adds them to the Gui Tree. If the *Ripper* encounters the same window again, the algorithm knows not to reverse engineer the window again. However, there are certain windows in GUI applications that do not have fixed titles. For example, the *Main* window of MS Word always shows the file-name of the currently opened document. By default, if the *Ripper* encounters such dynamically titled windows multiple times during its traversal of the application, it adds them all to the GUI Tree as different windows, and proceeds to rip each one separately. Human input guides the *Ripper* algorithm in such a situation.

Second, the application being ripped may have certain windows or widgets that interfere with the *Ripper*'s algorithm, e.g., by closing the application or invoking

¹⁹Standard GUI terminology; see detailed explanations at msdn.microsoft.com/library/en-us/vbcon/html/vbtskdisplayingmodelessform.asp and documents.wolfram.com/v4/AddOns/JLink/1.2.7.3.html.

an external application such as a browser that is not a part of the AUT. Additionally, certain widgets may cause undesirable side effects, e.g., sending large files to the printer. Human testers must enumerate these problematic widgets as input to the *Ripper* called the *ignore-list*. The *Ripper* also needs to know *a priori* about widgets which close a GUI window. These widgets, called *terminal* widgets—such as OK, Cancel, and Dismiss—are identified by the human tester and provided as input to the *Ripper* as the *terminal-list*. The *Ripper* uses the ignore-list and terminal-list to identify and handle those widgets as special cases. Ignored widgets are never exercised. Terminal widgets are exercised only when the *Ripper* intends to close a GUI window, only after completely extracting a window's structure.

Third, human input specifies an *initial state* for the AUT. The *Ripper* launches an application before identifying its top-level window(s) and proceeding to exercise its individual widgets. The application's initial state must be configured *a priori* by human input. Importantly, this choice of initial state can affect the components available in the GUI and detectable by the *Ripper*. For example, if Microsoft Notepad is launched with an empty document, the Edit->Copy menu item is inactive.

Fourth, during the ripping process, the *Ripper* may encounter text fields such as FileDialog entries, URL fields, or other similar inputs. The human tester can configure the *Ripper* with values to supply into these fields in a *text-entry-list*. For example, a GUI requiring a login and password can obtain this information when needed if specified in the text-entry-list. Based on the *text-entry-list*, the inputs to the application may trigger different, and more exhaustive, structural behavior.

All manual inputs, including the ignore-list, terminal-list and text-entry-list are provided as inputs to the *Ripper* in a single *configuration file*. The application's initial state is dictated by application arguments, which are specified when launching the *Ripper*, and sometimes application support files, which must be maintained externally.

At the end of the ripping process, the *Ripper* stores the extracted structural information about the GUI in a data structure called *GUI Tree*, in XML format. Figure 2a shows the GUI of a simple Java application. The application has two GUI windows, where the Exit button of the root window invokes the child window. Figure 2b shows the corresponding Gui Tree. For readability, only a subset of the extracted attributes of the GUI windows is shown.

A fundamental design decision for the *Ripper* was to shield other components of the GUITAR toolchain (*Graph Converter and Test Case Generator*) from platform-specific intricacies. Once an AUT has been ripped and a GUI Tree obtained, GUITAR does not need to interact with the AUT for generating test cases. Subsequent steps extract information from the GUI Tree to automatically generate appropriate graph models and test cases.

3.1.2 Graph converter

The *Graph Converter* provides a platform-independent framework to convert the Gui Tree model, output by the *Ripper*, into a graph representing relationships between events in the GUI of the application. The framework provides support for processing the input GUI Tree and generating a graph which is subsequently used for test case generation.

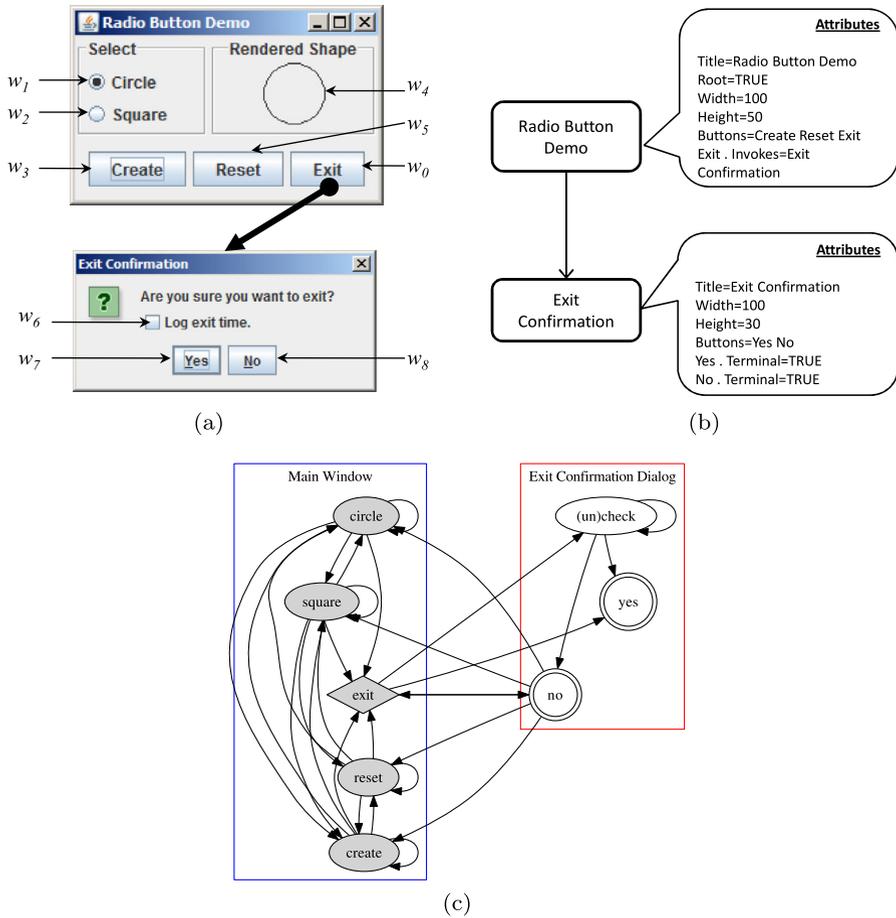


Fig. 2 GUI Tree and EFG of a simple RadioButton application; (a) shows a small GUI application with two windows; (b) shows its GUI Tree with 2 nodes and a subset of attributes; (c) shows the corresponding EFG

The default event model in GUITAR is the event-flow graph (EFG) (Memon et al. 2001b; Memon 2007). An EFG is a directed graph representing all possible event interactions on a GUI. Each node in an EFG represents a GUI event (e.g., click-on-Create, click-on-OK). An edge from node *v* to node *w* represents a *follows* relationship between *v* and *w*, indicating that event *w* can be performed immediately after event *v*. An EFG is analogous to a control-flow graph, in which vertices represent program statements and edges represent execution flows between the statements.

The construction of the EFG is based on the identification of modal and modeless windows. Recall that the *Ripper* classifies GUI events while extracting widget properties and encodes this information in the GUI Tree. The *Graph Converter* leverages this information while constructing the EFG. *Restricted-focus events* open modal windows. *Unrestricted-focus events* open modeless windows. *Terminal events* close modal windows. *Expand events* are used to reveal hidden sub-components (e.g.,

sub-menu items or tab elements). *System-interaction events* are not used to manipulate the structure of the GUI; rather, they interact with the underlying software to perform some actions.

The *Graph Converter* uses the above classification to compute the `follows` set for each event, which in turn is used to create edges of the EFG. We have presented the details of how to compute the `follows` set in our previous work (Memon et al. 2001b). Figure 2c shows the EFG for the application seen in Fig. 2a. In this EFG, there is an edge from *circle* to *circle* because a user can execute *circle* in succession; however, there is no edge from *circle* to *yes* because a user cannot execute *yes* after *circle*. The EFG has a set of *initial events*, shaded in Fig. 2c, which can be executed in the GUI's initial state. We show *exit*, a window-opening event, using a diamond; *yes* and *no*, both window-termination events, using the double-circle shape; and the remaining system-interaction events using ovals.

An *Event Model Converter* is similar to the *Graph Converter*, except that it transforms from one event model to another. GUITAR provides some built-in *Event Model Converters* to transform from the EFG to its variant models such as the event-interaction graph (Memon et al. 2001b), event-semantic interaction graph (Yuan et al. 2011) and probabilistic event-flow graph (Brooks and Memon 2007). Users of GUITAR can extend model converters to work with their own models and support tools based on these models.

3.1.3 Test case generator

The *Test Case Generator* is the next tool in the toolchain. Like the *Graph Converter*, this tool is also platform-independent. A *Test Case Generator* automatically generates test cases based on the graph model output from the *Graph Converter*. Although the tester is free to implement any algorithm for test generation, our current *Test Case Generator* framework provides support for taking, as input, the *Graph Converter's* output graph model and performing specified graph traversal algorithms on the model to automatically generate test cases. Hence, the GUI test generation problem is reduced to a problem of graph traversal.

The *Test Case Generator* framework provides three core features that may be used by a tester to implement a specific test generator. First, depending on the model exploration strategies desired, various test case generators may be built around a single graph model. In theory, a GUI test case can be of any length, possibly infinite, as a single widget can be clicked an infinite number of times. With multiple widgets on a GUI, the possible combinations can easily grow very large. Similarly, a GUI test suite can be of any size, possibly infinite. One can construct a test suite containing test cases of all lengths.

Second, the *Test Case Generator* generates values for event parameters if required (such as adding values for text-input fields by reading text inputs from the configuration file). Third, the *Test Case Generator* framework inserts “connecting” events in the test case to make it executable on the real GUI. Connecting events may be added to the test case $e_1 \rightarrow e_2$, to create $c_1 \rightarrow e_1 \rightarrow c_2 \rightarrow e_2$. This insertion makes e_1 accessible from the initial state of the GUI, and also makes e_2 accessible from e_1 . Here, the added event c_1 is called a “prefix” event.

3.1.4 Replayer

The *Replayer* provides a framework for executing a test case automatically on the AUT. The tool also provides hooks for observing and recording the AUT during test case replay. The *Replayer* takes as input an executable test case, the Gui Tree and the graph model, launching the application with the same initial state used during ripping. For each event in the test case, the *Replayer* uses the information in the GUI Tree and the graph model to identify the GUI window and widget on which the event needs to be executed. The tool invokes the event on the identified widget. When all events have executed, the *Replayer* closes all open windows and shuts the application down.

As was the case with the *Ripper*, the human tester may pre-configure the *Replayer* with values to supply into input fields using the *text-entry-list*. This list is provided as input to the *Replayer* using a *configuration file*. The tester may also use *Replayer*-provided hooks called *Test Monitors* to implement plugin-specific steps to observe and record the state of the GUI. For example, plugins may record some aspect of the state of the application during execution. This information can serve to validate the result of the executed test case or by directly comparing it with a previously recorded state. Test Monitors can also collect any other available run-time information, such as data supporting event-specific code coverage analysis.

The *Replayer* is a platform-dependent component. Plugins need to be implemented specific to the platform since they interact directly with the AUT.

3.1.5 Oracle verifier

The *Oracle Verifier* provides mechanisms to determine whether a GUI executed correctly for a test case. In addition to a sequence of events to be executed, a test designer must decide both what to assert and when or how often to check an assertion, e.g., after each event in a test case or after the entire test case completes execution. Variations of these two factors significantly impact the fault-detection ability and cost of the execution and maintenance of a GUI test case. Currently, we support two Oracle Verifier implementations with GUITAR: the *CrashVerifier* for reporting crashes (demonstrated in the Case Study 0) and the *StateVerifier* for matching output GUI states across different test case executions.

In general, GUITAR supports assertions of GUI state both during (through Test Monitor extensions) and after test case execution (as in the *CrashVerifier* and *StateVerifier*). The types of oracles supported by GUITAR are limited only by the ability to extract and verify GUI attributes during or after a test. Further analysis of GUI oracle choices with GUITAR and their effects on fault detection is outside the scope of this paper.

3.2 JFC toolchain

We have used the GUITAR framework to implement a set of tools for testing applications written with the Java Foundation Classes (JFC). The tools form a toolchain which can be used to automatically test a JFC application. In this section, we describe tools implemented for the JFC toolchain.

3.2.1 JFCGUIRipper

JFCGUIRipper is a plugin based on the *Ripper* framework. As a *Ripper*, the tool is capable of extracting GUI structural information from JFC-based GUI applications.

The JFCGUIRipper uses reflection to interact with GUI widgets. Specifically, the tool interacts with all widgets that support the Java Accessibility framework (all standard Swing and AWT widgets support this framework). The default GUI events such as left click on clickable widgets (e.g., button, checkbox) and type-in-text on editable widgets (e.g., text box) supported by the Java Accessibility framework are recognized and captured. In addition, the *Ripper* algorithm needs specific guidance to discover relationships among GUI events and widgets. This information is encoded in the tool's adapters. JFCGUIRipper includes custom adapters for ripping `JTabbedPane` and `JTree` widgets correctly. The ripping process for all other widgets follows GUITAR's generic *Ripper* algorithm previously described.

Given the default implementation of JFCGUIRipper, some non-standard widgets and their properties may be missed during ripping. In order to interact with and extract properties from these additional widgets correctly, application-specific implementation beyond the existing JFCGUIRipper is required. These additional extensions yield a more accurate GUI Tree and EFG. In a later case study (Sect. 5.4), we describe extensions to the JFCGUIRipper supporting the extraction of a more detailed GUI structure for a specific application.

3.2.2 EFGConverter and sequenceLength generator

The JFC toolchain includes a single *Graph Converter*, EFGConverter (Memon 2007), based on the *Graph Converter* framework. This platform-independent plugin is usable with the JFC toolchain as well as any other GUITAR toolchain. The EFGConverter converts the GUI Tree generated by the JFCGUIRipper into an EFG, based on the event-flow model.

SequenceLength Generator is the *Test Case Generator* of the JFC toolchain, and is likewise platform-independent and based on the *Test Case Generator* framework. This tool generates all possible test cases of a specified length (L). The resulting suite contains one test case for each event sequence of length L in the input EFG. For $L = 1$, the resulting suite covers every event in the EFG, for $L = 2$ the suite covers every pair of events occurring in the EFG, and so on for longer lengths. Each individual test case covers one sequence of length L , with a prefix applied to the sequence to make it accessible from the application's initial state. Note that this prefix means the actual length of test cases may in fact exceed the value of L .

3.2.3 JFCGUIReplayer

The JFC toolchain includes a test case *Replayer* based on the *Replayer* framework called JFCGUIReplayer. The JFCGUIReplayer is platform-dependent, being able to execute test cases on a JFC-based application.

The JFCGUIReplayer implements Test Monitors to store the state of the GUI after executing each event in the test case as a XML file. As a *Replayer*, the JFCGUIReplayer executes a test case in isolation rather than in the exploratory fashion of the

Ripper. The observed state may therefore be unique, having never occurred during the ripping process. This XML GUI state file can be analyzed to determine if a test case passed. The analysis may include comparing the state with the state stored from a previous execution, for example, during regression testing.

The JFCGUIReplayer follows the generic *Replayer* algorithm presented earlier to execute each test case. Java applications in particular have a high cost associated with launching the application to ensure an identical initial state. This cost leads to the JFCGUIReplayer's execution time dominating the compute cost in this case study, and this result is typical for GUITAR workflows in other platforms.

A simple shell script, acting as a test oracle, examines log files generated by JFCGUIReplayer. In addition to a simple CrashVerifier oracle, the script also detects exceptions and error strings from execution logs, which human testers can then manually inspect.

3.3 Case study 0

This section describes a simple workflow, called the *standard workflow*, that uses the JFC toolchain (Sect. 3.2)—JFCGUIRipper, EFGConverter, SequenceLength Generator, JFCGUIReplayer—to “crash test” two JFC applications. Figure 1 shows the JFC standard workflow. Ovals represent processes and boxes represent testing artifacts and results.

The case study was executed on the ArgoUML and JabRef JFC applications. Table 2—*Lines of code, Windows, and Widgets*²⁰—shows that these are non-trivial applications.

The first step was executing the JFCGUIRipper. This step first requires manually specifying the *ignored* and *terminal* widgets. Certain widgets have to be *ignored* and some marked as *terminal* for a smooth and accurate ripping process (Sect. 3.2.1). To determine the ignored and terminal set of events, we first ran the *Ripper* with no configuration. The generated GUI Tree was then processed (with an XML parsing script) for a count of windows and widgets. If the *Ripper* got stuck, we considered ignoring the last successfully executed event. If the *Ripper* completed, but the counts of GUI windows were intuitively lower than expected, we considered adding the last executed event to the set of terminal widgets. For ArgoUML, the application's “Print” dialog was ignored, by this process, to avoid physical printing during ripping. Also, the “Help” dialog, which contributed very little to coverage, was ignored. For JabRef, in addition to “Help”, certain widgets that induced dynamic changes in titles of GUI windows, which prevented the *Ripper* from correctly identifying GUI windows, were ignored. This problem is an artifact of the *Ripper*'s approach of identifying windows and widgets detailed earlier.

The set of terminal widgets remains fairly consistent across the two applications. The final configurations of both applications marked widgets with labels such as “close”, “open”, “save”, “ok”, “cancel”, “quit”, “yes”, “no”, “save”, and “exit”, with some application-specific differences, such as the label “Close database” for JabRef.

²⁰The numbers reported here are for all available widgets, including non-interactable widgets (e.g., labels, pictures) and invisible widgets (e.g., layout panels, tab panels).

Table 2 Case study 0: Result summary

	ArgoUML	JabRef	
Ripping	Lines of code	69,954	44,522
	Windows	30	40
	Widgets	1,548	1,285
	Ignored	2	6
	Terminal	13	14
	Time (s)	231	431
Model conversion	Nodes	328	376
	Edges	4,468	15,562
	Time (s)	4	5
Test case generation	Test cases	4,468	15,562
	Avg length	2.82	4.80
	Time (s)	213	875
Replaying	Statement coverage	22.45 %	29.12 %
	Branch coverage	10.31 %	12.04 %
	Log size (GB)	8.4	185.0
	Fault detected	3	4
	False positives	6	8
	Time (hours)	309	1,204

Because the configuration of terminal widgets in GUITAR supports matching on any subset of widget properties (including just the label), an existing set of common terminal widget labels could provide a default terminal-list for GUITAR.

Note that the time reported for *Ripper* execution includes only the execution time of the *Ripper* under its final configuration. This time does not include the time required for manual configuration. We did not measure the time required for *Ripper* configuration, as the testers in this case study (the authors of this paper) do not represent typical users of GUITAR. Because we started from empty configurations of ignored and terminal widgets for each application, the number of ignored widgets (2 for ArgoUML and 6 for JabRef) and terminal widgets (13 for ArgoUML and 14 for JabRef) gives an imprecise indication of the manual effort required for *Ripper* configuration. The cost of this manual configuration includes a number of iterations (approximately one per each ignored or terminal widget) of running only the automated *Ripper*, then manually checking for any issues and updating the configuration.

After the *Ripper* was sufficiently configured to generate a reasonable GUI Tree, the EFGConverter automatically generated the EFG from the GUI Tree. Table 2 shows the number of nodes and edges in the EFG obtained for ArgoUML and JabRef.

Using the EFG, the SequenceLength Generator generated test cases covering all event sequences of length 2. The total number of test cases and the average test case length in each suite is reported in Table 2. Recall that the average length can vary from the length 2 because the *length* parameter determines the length of event sequences

chosen from the EFG. An initial set of events may be prefixed to the generated test case to make the first event reachable (Sect. 3.2.2). The first three steps—ripping, graph conversion and test case generation—were executed on a machine with 1 GB main memory and 2 GHz single core CPU running Redhat Linux Enterprise Linux 5.

Finally, the JFCGUIReplayer executed all of the generated test cases, completely unattended, on a homogeneous cluster of 120 machines with the above configuration. Table 2 shows the statement, branch coverage and faults detected during replay. The total time taken to execute the test suite is also reported. This time is the sum of execution time for all test cases. As expected, test replay time dominated the total computation time of this study.

Logs from the JFCGUIReplayer were analyzed using scripts, acting as simple oracles, to detect faults (Sect. 3.2.3). Under manual inspection of script output, some exceptions were found to have occurred due to inactive widgets or due to expected widgets not being found in the GUI during test case execution. On closer manual analysis, these were found to have occurred due to limitations of the *Ripper* in extracting a completely accurate GUI Tree (Sect. 3.1.1). We classified these specific faults as *false positives* and they are not real faults in the AUT. ArgoUML encountered 6 such instances, and JabRef encountered 8.

Table 3 reports faults detected for each application. We detected 7 crashes (3 in ArgoUML and 4 in JabRef) as shown in Table 3. These crashes had never been reported before to application developers. We reported each fault, and developers have fixed the faults in subsequent versions of their applications. The bugs found reinforce the importance of considering events in many possible contexts rather than a single event execution. Most bugs we found would not have been found by a single event execution. Critically, the EFG provides a known path to every captured event from the initial state of the application, allowing coverage of each desired sequence and resulting in longer, more effective test cases.

Figures 3 and 4 show a test case which led to the discovery of a specific fault in JabRef, identified as fault JR_2 . Figure 3 shows the complete EFG obtained for JabRef. The large nodes highlight the event-sequence leading to the fault, with the last event triggering the fault. Figure 4 depicts the test case's execution on JabRef. The events e_1 , e_2 and e_3 were prefix events required to reach the generated length-2 sequence of $e_4 \rightarrow e_5$.

Although our goal was coverage of length-2 event interactions, this test case required a length of 5 to cover the desired sequence. This systematic generation of longer sequences makes the test suite more effective while being computationally tractable. By covering all length-2 interactions, we were able to find bugs in less-traversed (and likely less-tested) portions of the software.

Several things stand out in the extracted data. First, the test suites have a high number of possible test cases even when covering sequences of only length 2. This number increases exponentially as the covered sequence length increases, which magnifies the importance of test case automation and also emphasizes that most possible test cases are necessarily overlooked by exploratory testing. With a model-based technique, we can make stronger arguments for test adequacy, but with so many test cases to consider, tool scalability becomes important. We consider these problems in a case study of continuous integration testing later in this paper (Sect. 5.2).

Table 3 Case study 0: Faults detected*

Fault ID	Summary	Test case
<i>AU</i> ₁	<code>FileNotFoundException</code> with invalid input file name for Export Graphic	Expand 'File' menu → Click 'Export Graphic' submenu → Enter an invalid file name → Click 'Save'
<i>AU</i> ₂	<code>FileNotFoundException</code> with invalid input file name for Export All Graphic	Expand 'File' menu → Click 'Export All Graphics' submenu → Enter an invalid file name → Click 'Save'
<i>AU</i> ₃	An inappropriate exception trace printed out when deleting object with a blank document	Expand 'Edit' menu → Click 'Delete from Model' submenu
<i>JR</i> ₁	<code>FileNotFoundException</code> with an non-existing Journal abbreviation file	Expand 'Option' menu → Click 'Manage journal abbreviation' submenu → Enter an invalid New file name → Click 'OK'
<i>JR</i> ₂	<code>MalformedURLException</code> with an invalid Journal abbreviation download URL	Expand the 'Option' menu → Open 'Journal abbreviation' windows → Click 'Download' button → Enter an invalid URL → Click 'OK'
<i>JR</i> ₃	<code>NullPointerException</code> with invalid import folder name	Expand 'Option' menu → Click 'Manage custom imports' submenu → Click 'Add from folder' → Enter a non-existing folder path → Click 'Cancel'
<i>JR</i> ₄	<code>ZipException</code> with invalid zip file name	Expand 'Option' menu → Click 'Manage custom imports' submenu → Click 'Add from jar' → Enter a non-existing zip file name → Click 'Select a Zip-archive'

*A full fault report is available at <http://www.cs.umd.edu/~baonn/projects/guitar/bugs>

The code coverage numbers reported here may seem low. The seemingly low numbers can be attributed to several factors, including:

1. As a form of system-level testing, GUI testing can only exercise parts of the code exposed to the GUI as executed by the *Ripper*. We have executed the *Ripper* in a single environment with specific inputs. To improve coverage, these factors can be varied by controlling the *Ripper*. Modifying inputs to the *Ripper*, e.g., for text-input fields, could lead to increased coverage of widgets and code.
2. The *Ripper* may miss GUI elements as a result of execution within a single application instance. To improve coverage, application-specific ignored and terminal components could be configured more precisely. We have not considered deeper application-specific configuration in this case study.
3. We have only considered test cases covering sequences of length 2. To improve coverage, one could add more test cases, or focus on generating *better* test cases, according to event coverage (e.g., sequences of length 3) or any other criteria configurable by a test case generator. GUITAR's support for customized test case generation is discussed in Sect. 5.1.

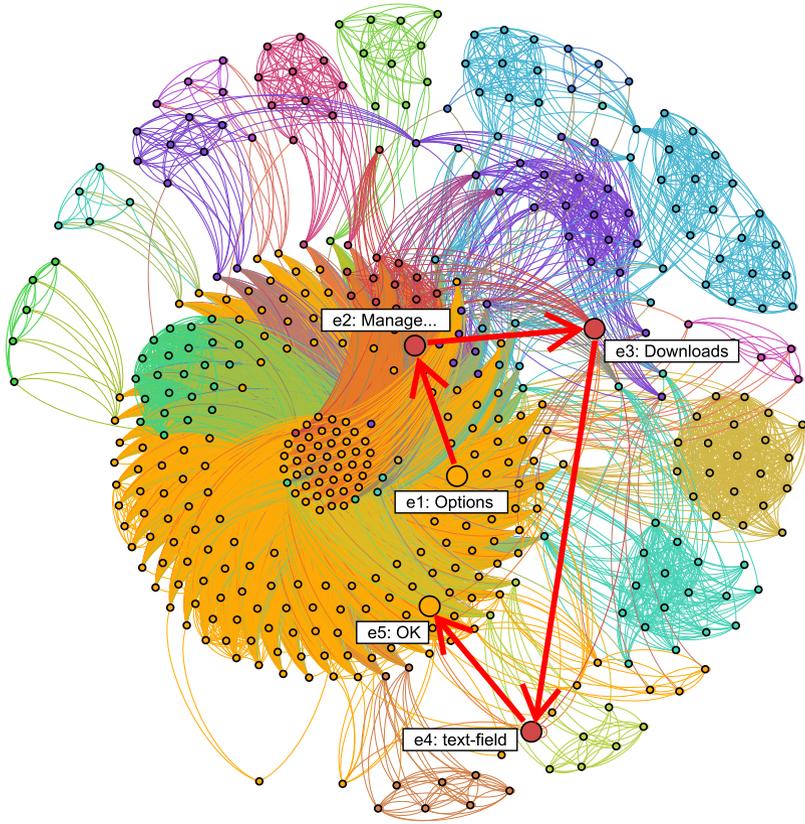


Fig. 3 EFG for JabRef with one event sequence triggering a fault

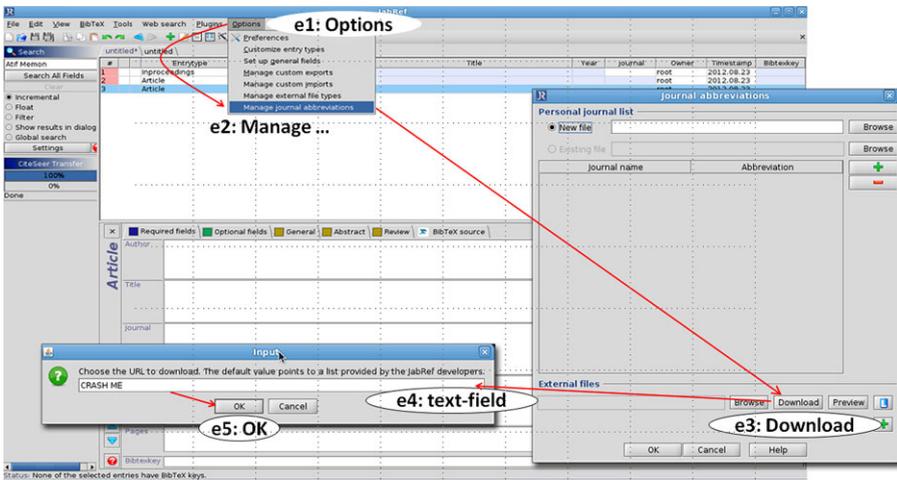


Fig. 4 The fault revealing event sequence of *JR*₂

4. GUITAR tools may not know how to interact with all encountered widgets and events. We expand on GUITAR's ability to consider additional widgets and events in Sect. 5.4.

GUITAR may also not appear very *efficient*. From Case Study 0, two aspects of GUITAR appear resource intensive—the total amount of time consumed to execute test cases and the total amount of artifacts generated (see Table 2). Extensive resource requirements are a general problem of such model-based testing approaches. The wall-clock time to execute test cases can be reduced, to a practical turnaround time, by using a cluster of inexpensive and optionally virtual machines.

In summary, Case Study 0 leads to the following observations:

1. The model-based workflow of GUITAR leads to the generation of useful test cases uncovering real faults.
2. The *Ripper*'s window and widget identification process has difficulties when window titles change.
3. The *Ripper* executes on an application's GUI from a single initial application state. This initial state may cause some widgets to be inaccessible and not to be extracted into the GUI Tree. In addition, the *Ripper* follows only one order of traversing the GUI. This may cause certain widgets to remain inaccessible and also not to be extracted into the GUI Tree.
4. The manual configuration process can be addressed by a systematic manual procedure, but GUITAR offers no direct support for quick validation of GUI Trees.
5. GUITAR's ability to match against any subset of widget properties when identifying widgets allows the configuration of terminal widgets across applications to leverage applications' tendency to reuse common labels for window-closing widgets.

4 Supporting extension: the GUITAR architecture

In this section we describe the *component-based* design (Alfaro and Henzinger 2001) of GUITAR, emphasizing the framework's support for plugins and extensions which enable more complex testing workflows and tool use cases.

GUITAR's components can be grouped into: '*core*', '*tools*' and '*plugin*' components. '*core*' components provide global services. '*tool*' components provide building blocks for individual tools. '*plugin*' components add customized features to tools. Users may use tools independently or integrate them into *toolchains* for a customized workflow supporting specific testing activities.

GUITAR is implemented in Java. Each GUITAR component has two layers to improve flexibility and extensibility. The *abstract layer* defines an API for communicating with other components using abstract classes and interfaces. The *implementation layer* provides low-level implementation details for the component. This separation makes components interchangeable, such that replacing one component does not interfere with other components of the framework. We now describe each component in detail.

4.1 Model core

The central component in GUITAR is the *Model core*. This component defines the following data structures common to all GUITAR components:

- The *GUI Structure* represents the GUI hierarchy, containing a set of all GUI windows in the application. Each window contains GUI components with their properties and associated values. In GUITAR, a GUI Structure can be used to represent either the static structure of the entire GUI or a dynamic GUI state as observed at a particular time. The GUI Structure is used to store the GUI Trees generated by the *Ripper* as well as the GUI states captured by the *Replayer*.
- An *Event Model* represents the relationships between events in the GUI Structure. The Event Model is a directed graph with nodes representing events and edges representing relationships between events (e.g., the `follows` relationships in the EFG). *Test Case Generators* use an *Event Model* to systematically generate test cases.
- The *Test Case* structure represents a sequence of GUI events which can be performed one after another on the application from its initial state. A test case can optionally contain a sequence of *GUI Structure* objects representing the expected state of the GUI after each event as a form of an assertion.

All GUITAR components interact with one another using the common data structures defined in the Model Core.

4.2 Platform-specific components

GUITAR's design emphasizes platform independence as much as possible. The *Graph Converter* and *Test Case Generator* are platform-independent (see Sect. 3). However, the *Ripper* and *Replayer* require platform-specific implementation in some GUITAR components which interact directly with the GUI.

To enable the interactions between platform-specific and platform-independent components, we provide an intermediate component called *Executor*. The Executor consists of two sub-components: (1) The *Native GUI Automation* component is a platform-specific library such as Java Accessibility or Selenium Web Driver. This component directly interacts with the GUI. (2) The *Executor Bridge* component communicates with the Native GUI Automation component to support the platform-independent Executor API. This API works as a contract between the platform-specific details of the GUI Automation library and the high-level, platform-independent models defined in the Model Core. The Executor API interfaces with all other GUITAR components, so that once the Executor API is implemented, the platform-specific components of the Executor can communicate with the rest of GUITAR in a platform-independent way.

The Executor API includes four interfaces:

- `GApplication`²¹: represents a GUI application and methods to initialize applications, such as starting and terminating the GUI and accessing window handlers.

²¹The prefix “G” indicates that a component is a GUITAR abstract class.

- `GWindow`: represents a GUI window and methods to access window properties.
- `GComponent`: represents a GUI component (e.g., a widget) and methods to access component properties.
- `GEvent`: represents an event type such as left-click, right-click, and text entry. A `GEvent` paired with the `GComponent` represents a specific GUI event on a GUI component (e.g., a left-click on the OK button).

The first three interfaces provide access to the content of the GUI such as the GUI hierarchy and GUI properties. `GEvent` provides functionality to interact with the GUI. Section 5.5 will provide a case study describing ways to implement the Executor API for a specific platform. The Executor plays an important role in GUITAR, replacing the need for manual interaction with GUIs to enable the use of much larger test suites.

GUITAR does not impose any restrictions on the types of applications to be tested. However, a specific Executor implementation will only work on applications of a certain kind, due to the Executor's dependence on GUI automation. For this reason, we refer to GUITAR as supporting *platforms* of applications which can each be accessed by a specific Executor implementation.

While we do not explore such extensions within this paper, an existing Executor can support entirely new tools requiring GUI automation and model-based considerations, such as a capture tool or alternative reverse engineering tool.

GUITAR contains two instances of the *Executor*: the *Ripper* and the *Replayer*. These concrete instances implement two different automation strategies on the GUI as described below.

4.2.1 The ripper

The default behavior of the *Ripper* (see Sect. 3.1.1) can be customized using plugins called Ripper Adapters. A Ripper Adapter provides a hook for plugins to customize the ripping step. The *Ripper* executes a plugin's functionality before and after each ripping step, allowing the plugin to override the default GUI traversal strategy of the *Ripper*.

Developers can implement a specific Ripper Adapter by extending the abstract class `GRipperAdapter`, which has two pertinent methods:

- `isProcessed`: specifies which components should be handled by this Ripper Adapter.
- `ripComponent`: specifies how the *Ripper* should proceed with handling (e.g., interacting with and extracting properties from) the identified components.

For example, an adapter called `IgnoreComponentAdapter` implements the capability to ignore components. This adapter is used in the `JFCGUIRipper` of Case Study 0. The implementation overrides the *Ripper*'s handling of components specified in the configuration file so that the *Ripper* skips these components. Section 5.4 will provide a comprehensive example, where we use a custom Ripper Adapter to enable the `JFCGUIRipper` to handle customized GUI components.

Table 4 Case Studies 1–5 using GUITAR

ID	Scenario	Required extensions
1	Alter the standard workflow to randomize the generation of test cases	Custom GUITAR module based on TestCaseGeneration Core
2	Distribute test case replay across a cluster of machines	Test harness scripts for deploying GUITAR tools. No change to GUITAR code
3	Collect statement coverage of individual GUI events	Custom test monitor based on <i>Replayer</i>
4	Add support for an additional widget and event	Custom widget type and custom event type in Platform Model
5	Develop a new toolchain to support the automated testing of a new platform	Custom Executor for the new platform

4.2.2 The replayer

The default behavior of the *Replayer* (see Sect. 3.1.4) can be customized using plugins called Test Monitors. A Test Monitor injects customized monitoring steps during test case execution. Test Monitors extend the `GTestMonitor` interface with four methods which are invoked at specific points during test case execution:

- `init`: invoked before any event is executed.
- `beforeStep`: invoked before an individual event is executed. It takes a `GTestStepEventArgs` object as argument to pass in any step-specific data (e.g., event ID).
- `afterStep`: invoked after an individual event is executed. It also takes a `GTestStepEventArgs` object as an argument.
- `term`: invoked after all events are executed.

For example, the `JFCGUIReplayer` implements a `StateMonitor` to capture GUI states during test case execution. In this monitor, the `afterStep` method records GUI states after the execution of each test step. Those states are exported as GUI Structure XML files that can be examined to determine test results. Section 5.3 will describe another Test M.

5 Extending GUITAR: case studies 1–5

With an understanding of GUITAR's basic functionality (Sect. 3) and support for extensions (Sect. 4), we now consider five customized workflows using GUITAR as shown (Table 4). In each case, we describe a testing scenario and develop a corresponding workflow supported by GUITAR tools. We describe the development of tools beyond the JFC standard workflow and provide supporting artifacts. These case studies highlight the flexible nature of GUITAR and its utility in GUI testing.

Table 5 Case study 1 results

		ArgoUML	JabRef
Test case generation	Test cases	4,468	15,562
	Avg Length	4.36	5.50
	Time (s)	1,953	2,482
Replaying	Statement coverage	24.72 %	28.70 %
	Branch coverage	11.74 %	11.59 %
	Fault detected	2	4
	Time (h)	431	1,679

5.1 Case study 1: a custom test case generator

The SequenceLength Generator of Case Study 0 provides nice guarantees of event coverage in the EFG. However, using longer sequence lengths could discover more interesting bugs. The number of maximum possible test cases increases exponentially with an increase in the length of test cases. Therefore, a tester might consider random sampling as an alternate method for test case generation. In this case study, we propose an alternative test case generator, *RandomSequenceLength Generator*, implemented as a custom plugin to the *Test Case Generator* framework.

The RandomSequenceLength Generator takes two arguments, L and M , and automatically generates test cases covering a randomly sampled sequence (without replacement) from all possible sequences of length 1 to L to build a suite of M unique test cases. Aside from the selection of the covered sequence, the RandomSequenceLength Generator functions exactly as the SequenceLength Generator, prefixing a path to an initial event, if necessary.

The sampling strategy in the RandomSequenceLength Generator is different from those in the random walk testing tools discussed in Sect. 2.2. These tools typically blindly navigate the GUI and perform events on encountered widgets without considering the overall GUI input space. In contrast, the RandomSequenceLength Generator is guided by the EFG model of the GUI. By uniformly sampling from a known space, the events in the generated test suite are equally distributed. In addition, because test suite generation can be done before execution, the tester can perform further analysis before executing any test cases.

5.1.1 Results

We applied the RandomSequenceLength Generator to the existing EFG files for ArgoUML and JabRef with parameters of $L = 4$ and M equal to the number of test cases in each application's complete SequenceLength $L = 2$ suite from Case Study 0. Table 5 shows the results of Case Study 1, as compared to the results of Case Study 0 from Table 2.

From the raw data, we notice an increase for both applications in average test case length, as a result of sampling from sequences of longer lengths. Test suite generation time, as measured by the time needed to generate the test suite, also rose for

both applications. This result is also expected. In the case of ArgoUML, code coverage measures increased; but we observed slight decreases in coverage for JabRef. Because of the randomness of the new test suite, we expect some differences in coverage. We did confirm that the coverage of the two case studies does not completely overlap.

We also take a deeper look into the defects found. We observe that not all of the defects from Case Study 0 are detected by the new test suite. This outcome is not surprising, because the new test suite (unlike the previous) does not guarantee coverage of all length-2 sequences.

The new suites, though, did uncover new faults by exercising different sequences sampled from longer sequence lengths. For example, in fault JR_5 , an `ArrayIndexOutOfBoundsException` occurs when attempting to add a new string constant to an already closed BibTeX file in JabRef. The test case revealing this fault consists of 6 events (e'_1 : *Expand BibTeX menu*; e'_2 : *Open 'String for BibTeX file' window*; e'_3 : *Close the BibTeX file*; e'_4 : *Open 'Add Input String' window*; e'_5 : *Enter some string to the text box*; e'_6 : *Click OK to add string*). This fault was not detected by the $L = 2$ suite of Case Study 0 because detection requires an interaction of the three events e'_3 , e'_5 , and e'_6 . Missing one of these events will not uncover the fault.

The developers of JabRef confirmed that the behavior tested by this sequence of closing the database then adding a string should be prohibited by making the *String for BibTeX file* window modal. This behavior was implied in their software's specification, but not enforced in the actual implementation, leading to the fault. This bug illustrates GUITAR's ability to find bugs based on execution. This capability complements existing specification-based GUI testing approaches (Silva et al. 2008; Chen et al. 2005).

In summary, Case Study 1 adds the following observations:

1. The GUITAR framework allows test case generation techniques to be implemented with no dependence on platform-specific code by the extension of one abstract class.
2. GUITAR supports randomly sampling from models, which allows a tradeoff between considered event interactions, test suite size, and the exhaustiveness of a test suite. This tradeoff gives the tester control over the scalability of the exponential cost of considering longer sequences.
3. GUITAR creates test cases from *AUT execution* which contrasts with testing from specification.

5.2 Case study 2: continuous integration testing

The previous two case studies have given rise to very large test suites. GUITAR offers automated replay of test cases, but execution of an entire suite constructed from a model-based coverage criteria would last for weeks on a single machine. Fortunately, the separation of logic amongst GUITAR's tools enables distribution of test case replay. In this case study, we demonstrate distributed continuous testing using GUITAR. This workflow results in a much quicker turnaround time than the standard workflow.

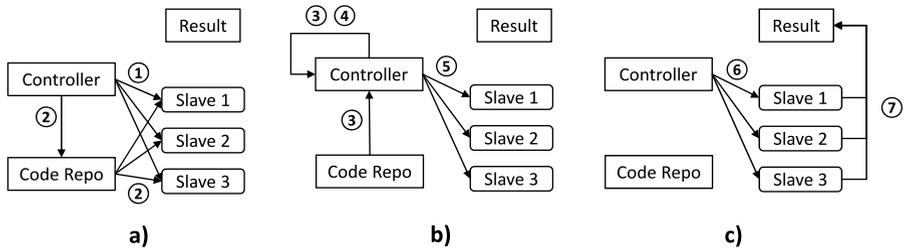


Fig. 5 Schematic representation of a continuous integration testing system using GUITAR. (a) Controller instantiates GUITAR and AUT on slaves, (b) generates test cases and transfers in batches to slaves, (c) triggers execution of test cases and receives archive results

To support this scenario's need for a quick turnaround of test case results, we develop a distributed workflow for GUITAR. We use the JFC standard toolchain to set up a testbed for Java applications which consists of four components: a *code repository*, a *controller machine*, a set of *slave machines* and a *result repository*. The testing process follows seven steps as marked in Fig. 5. The controller guides the distributed workflow as follows:

1. Initialize all slave machines with GUITAR tools
2. Compile from source and package as necessary the latest version of the application under test (AUT), e.g., from its trunk or specific branches
3. Generate GUI Tree and EFG on controller by running JFCGUIRipper and EFG-Converter
4. Generate test cases on controller using SequenceLength Generator
5. Distribute test cases in even batches to slave nodes
6. Trigger execution of test cases on slave machines and wait for completion
7. Wait for slave machines to post results directly to the result repository

Figure 5 shows a schematic representation of a distributed deployment of GUITAR. In a development lifecycle, this workflow can be triggered by code changes or on a periodic basis (e.g., daily or nightly builds). We integrated this workflow with a continuous build tool called Jenkins²² to guide the controller machine.

5.2.1 Results

We leveraged this distributed workflow for all case studies requiring the execution of large test suites in this paper. Development of the distributed workflow from execution of the JFC standard workflow on a single machine required no changes to the GUITAR code—only changes in the setup and deployment of tools. This flexibility exemplifies GUITAR's support for reuse of entire tools—and in this case the entire JFC toolchain—to support alternative workflows.

We note that steps 2–4 of our distributed workflow could be pre-processed and test cases reused from prior builds if developers do not expect the GUI of the application

²²<http://jenkins-ci.org>.

to change. However, if any aspect of the GUI changes, GUITAR's GUI tree, EFG, and resulting test cases may also change, and should be regenerated to capture the changes. The GUI Tree produced by the *Ripper* must always be validated to check if any newly introduced widgets or events should be ignored or marked as terminal, or if they require GUITAR extensions to be handled correctly by the *Ripper*.

Since many widget properties extracted by GUITAR and subsequently used to guide the *Ripper* and *Replayer* are environment-specific (e.g., position, width, height, etc. of widgets), the *Ripper* and *Replayer* tools should be run in the same environment. Many applications, even those implemented in cross-platform languages such as Java, render GUI structures differently on different architectures. In this case, we benefited from the use of a 120-node, homogeneous cluster. Since running this case study, we have also begun to experiment with the deployment and use of identical virtual machines for running distributed scenarios with GUITAR.

In summary, Case Study 2 of the distributed workflow adds the following observations:

1. Existing GUITAR toolchains require no modification for use in distributed testing scenarios.
2. Distributed testing with GUITAR still requires manual intervention upon GUI changes.
3. GUITAR toolchains require execution in a homogeneous environment.

5.3 Case study 3: measuring event-level statement coverage

In Case Studies 0 and 1 we have reported statement and branch coverage of each test suite as a whole. These values were measured during test case execution by Cobertura, an open-source, instrumentation-based library.²³ In this case study, we show GUITAR's support for collecting customized information during test case execution. Specifically, we consider the scenario of collecting statement coverage associated with individual test case events. This finer-grained coverage information, like similar information available at run-time, offers better support when locating faults. When a test fails, the tester can identify the last successfully executed event. Coverage attributed to the last event and recently executed events can more accurately direct the process of identifying the underlying fault which caused the test case to fail.

We implemented `CodeCoverageMonitor`, a Test Monitor which reports the underlying statements covered by individual test case events. The `CodeCoverageMonitor` extends the `GTestMonitor` interface described in Sect. 3.1.4. In the `init` method, we trigger coverage data collection after application load to measure initialization coverage. Then the `beforeStep` method resets the *coverage data object* state to empty before the execution of each event. The `afterStep` method saves a report after the event's execution finishes. Finally, the `term` method merges all event coverage information to provide a test case coverage report. This implementation ensures the creation of a fresh code coverage trace after launching the application and after executing each event in the test cases.

²³<http://cobertura.sourceforge.net>.

Table 6 Statement covered during test case execution

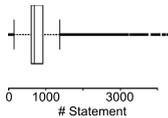
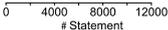
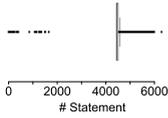
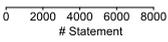
Application	Initialization	Event level	Test case level
ArgoUML	10,097		
JabRef	5,789		

Table 6 shows event-based code coverage for ArgoUML and JabRef. Column 2 shows code coverage for all code executed from application launch until prior to the first event of a test case. This coverage remains identical across all test case executions because the application starts with the same initial state for all test cases in our setup. This initial coverage provides an interesting measure of test suite adequacy. If the initial coverage represents a large fraction of an application's total lines of code, then the application may be easier to smoke-test. Column 3 shows the distribution of lines-of-code covered after executing each event. If executing an event tends to cover a small number of lines, this indicates that targeted test cases may be required to test all event-handling code in the application (Column 3). Column 4 shows the distribution of the lines of code covered after executing a complete test case. Similar to per-event coverage, the extent of line coverage by a complete test case indicates if targeted test cases may be required to increase code coverage.

Collectively, this coverage information characterizes an application. In the case of these two applications, we see that launching each application covers a large number of statements (Column 2). We also find that the distribution of statement coverage of individual events is fairly broad (Column 3). Some GUI events cover 6000 lines of code in JabRef, while others indicate little or no coverage.

Instrumentation with Cobertura does impose performance overhead on the application. To quantify this effect, in a small experiment, we executed 10 sequence length test cases with $L = 2$ on ArgoUML, both with and without instrumentation on a virtual machine with 1 GB RAM, 1.6 GHz 2-core CPU and running Fedora 12. The average test case execution time (excluding an initial *Replayer* wait time of 20 seconds) was 37 seconds with instrumentation and 33 seconds without instrumentation, for a little over 12 % increase in execution time. Cobertura also consumed an average of 68 seconds for post processing the coverage log after each test case execution completed.

In summary, Case Study 3 demonstrated GUITAR's ability to support data collection during test execution through a single interface called the Test Monitor.

5.4 Case study 4: custom components and event types

In many cases, the application under test may use custom or otherwise unsupported GUI components. Custom events, custom widget-specific properties, and custom im-

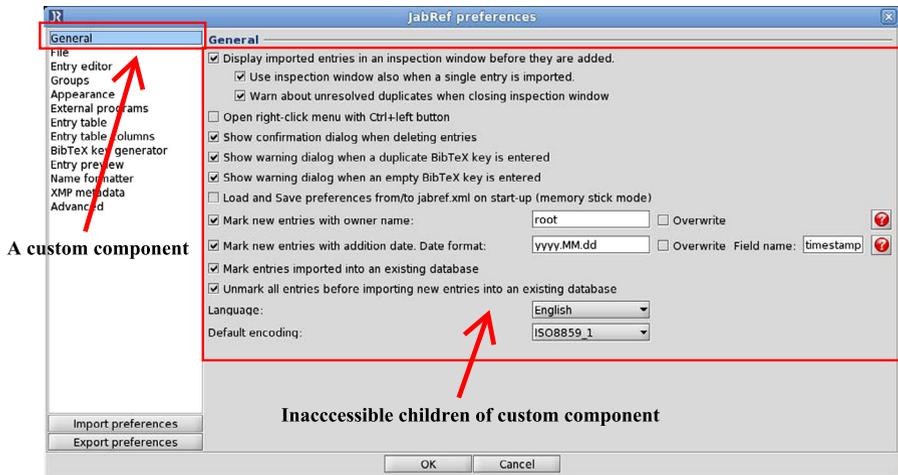


Fig. 6 Customized component in JabRef Preferences window

plementation can affect the *Ripper*'s ability to extract GUI widgets and their properties. To better gather properties and interact with such components, a custom extension of GUITAR is required. In this case study, we consider an extension of GUITAR which improves testing of JabRef using custom components and events.

5.4.1 Custom GUI components

When ripping an application, the *Ripper* delegates the ripping of custom components to *Ripper Adapters* (see Sect. 4.2.1). Each adapter directs the extraction of its corresponding components during ripping to make the GUI Tree richer, improving the accuracy of subsequent models and test cases.

JabRef uses a custom-developed GUI component called `GeneralTab`. This component improves the appearance of the `Preferences` window (see the left part of the window in Fig. 6). Because of the implementation of `GeneralTab`, GUITAR by default does not know how to discover its child components, such as the components revealed on the right-hand part of the window when an item is selected on the left side. By default, when a `GeneralTab` is selected, the corresponding GUI components revealed on the right-hand side of the window do not show up directly as children of the `GeneralTab` in the GUI Tree. This problem occurs because the implementation of `GeneralTab` creates a separate panel and explicitly moves the affected components to their new location. Without support for this custom component, the *Ripper* attempts to handle the component as a standard Java Tab, missing all of the components on the right-hand side.

We implemented `GeneralTabAdapter`, which follows the specific logic of `GeneralTab` to extract the previously missed components. When the *Ripper* encounters a `GeneralTab` object, this adapter automatically searches for the location of the `GeneralTab`'s children and redirects the ripping of the tab to the new locations, as appropriate.

Table 7 Custom components and event types

	Original	CC	CE	CC & CE
Component	1,285	1,709	1,285	1,709
Window	40	40	40	40
EFG node	376	483	380	489
EFG edge	15,562	38,555	15,680	40,355

CC = Customized Component; CE = Customized Event type

5.4.2 Custom event types

The GUITAR architecture manages GUI events separately from GUI components. GUITAR supports the implementation of customized *event types* for interacting with GUI components in custom ways. All event types in GUITAR extend the abstract `GEvent` class. There are two main methods in `GEvent` to implement:

- `isSupportedBy`: defines the class of components that support this event type.
- `performs`: defines what the event type actually performs on the components specified by the `isSupportedBy` method.

The existing JFC toolchain only supports a basic method to enter text at the beginning of a `JTextArea`. GUITAR can also be extended with a custom event to enter text at a *specified position* in the `JTextArea`. In this case, a `GEvent` class should provide an `isSupportedBy` method which recognizes `JTextArea` objects and a `performs` method which invokes the low-level methods of `JTextArea` to insert the input text at the specified position. This additional event complements the JFC toolchain's default text interaction of modifying the entire text of the component.

5.4.3 Results

By adding the custom component and event type described above, the quality of the GUI Tree and resulting event model obtained by GUITAR for JabRef improves. Table 7 contrasts the number of Components, Windows, EFG Nodes, and EFG Edges obtained by the *Ripper* when utilizing the custom component (CC), custom event (CE) and both (CC & CE), respectively. Recall that EFG nodes represent events and EFG edges represent inferred `follows` relationships.

As expected, including the custom component improves the number of components in the GUI tree by a large margin. By implementing `GeneralTabAdapter`, we gained over 400 previously missed GUI components. The *Ripper* already supported interaction with these child components, but previously missed them entirely because of not interacting properly with their parent. The new components led to over 100 new events in the EFG and increased the number of `follows` relationships by over a factor of 2. We see a slight improvement due to the custom event alone of only 4 EFG nodes, but this new event led to 118 new EFG edges. The combined effect of the custom event and component provides the most complete picture of the GUI input space to be tested.

Table 8 Mapping the internal GUI objects

	JFC platform	Web platform
GApplication	Application's main class	Selenium WebDriver object and the root page
GWindow	Java Window object	A Web page URL
GComponent	Java Component object	Selenium WebElement object

In our use of GUITAR tools, we consider the implementation of custom widgets and events to be a part of the manual *Ripper* configuration process, though this must be accomplished by implementation.

As mentioned previously, the manual process of configuring and evaluating *Ripper* output is problematic, and the potential need for custom components and events can initially increase the manual workload. However, we note that both types of extensions can potentially be reused across applications, to the same extent that applications reuse components and events. For example, if a developer keeps a library of custom widgets for use in many applications, a corresponding library providing GUITAR support for these components and their events would be just as portable. This observation also sets up our forthcoming case study of supporting entirely new platforms of applications with GUITAR in Sect. 5.5.

In summary, Case Study 4 finds:

1. Support for GUI components and events can greatly affect the GUITAR *Ripper* algorithm.
2. *Ripper* extensions for component and event support require implementation of classes for GUITAR, as well as code-level knowledge of both GUI classes and automation libraries.
3. GUI extensions in GUITAR may prove useful across applications, to the same extent that components and events themselves are reused.

5.5 Case study 5: GUI platform extension

The new toolchain applies GUITAR's algorithms to the testing of websites by leveraging browser automation. We compare the new Web toolchain to the JFC toolchain.

As mentioned in Sect. 5.4, the JFC toolchain leverages the Java Accessibility Framework to monitor and drive interaction with a JFC GUI. The Web platform uses Selenium WebDriver for the same purposes. Supporting a new platform requires extension of the *Executor API* of GUITAR, as described in Sect. 4. More precisely, extending *Executor* requires three steps:

Step 1: Mapping the platform's native objects to GUITAR's abstract objects. Each native GUI automation library (e.g., Java Accessibility Library, Selenium WebDriver) should have mechanisms for monitoring GUIs on the platform. This step involves identifying native objects in the platform which correspond to the abstract objects GApplication, GWindow and GComponent of the *Executor API*. Table 8 shows this mapping for both JFC and Web platforms. For example, in the JFC platform, GApplication only needs to know the tested application's main class.

Table 9 Accessing GUI component information

Interface	Method	Description	JFC platform	Web platform
GApplication	connect	Establish a connection with the application under test and start testing	Use reflection to find and invoke the main method in the main class	Use the <code>WebDriver</code> to start the browser and load the root page
	terminate	Disconnect with the application under test	Invoke <code>Java System.exit</code> method	Invoke quit method from the <code>WebDriver</code>
GWindow	getAllWindows	Get all windows currently available	Return the values of <code>Frame.getFrames</code>	Return all open pages
	isModal	Check if the window is modal or not	Invoke the <code>isModal</code> method in <code>Window</code>	Always returns <code>false</code>
GComponent	getContainer	Get the window's top level component	Return the window's <code>topJPanel</code> object	Return the top level <code>'body'</code> tags
	getTitle	Get title of component	Return text label or icon name of the <code>Component</code>	Return tag (e.g., <code>h1</code> , <code>img</code>) of the <code>WebElement</code>
	getClassVal	Get class of the component	Return class name of the <code>Component</code>	Return tag type of the <code>WebElement</code>
	getGUIProperties	Get all GUI properties and their value	Use Java reflection to find and invoke all bean methods of <code>Component</code>	Use the <code>getAttributes</code> method to get all attributes of <code>WebElement</code>

Table 10 Performing GUI events

GEvent	JFC platform		Web platform	
	Supported by	Implementation	Supported by	Implementation
Click	Components implementing the AccessibleAction interface	Invoke the doAccessibleAction method in AccessibleAction	The 'a', 'href' tags and the 'input' tags having type 'checkbox' or 'radio'	Invoke the click method in WebElement
EnterText	Components implementing the AccessibleEditableText interface	Invoke the setTextContents method in AccessibleEditableText	The 'input' tags having type 'text' and the 'textarea' tags	Invoke the sendKeys method in WebElement
Submit	Not available	Not available	The 'input' tags having type 'submit'	Invoke the submit method in WebElement

In the Web platform, a `WebDriver` instance and the URL of the site's root web page provides analogous information.

Step 2: Accessing GUI properties. This step requires implementing methods for `GApplication`, `GWindow`, and `GComponent` objects to access GUI functionality. Columns 2 and 3 in Table 9 detail the required methods, with reference implementations in Columns 4 and 5 for the corresponding platforms. As we can see, the platform-specific implementation details can be very different, as long as they provide the correct functionality to the Executor API. For example, in the JFC platform, the `connect` method call invokes the `main` method in the main class, which starts the GUI application. In the Web platform, a `WebDriver` object handles the connection by starting the browser, loading the root URL, and setting up the connection between the *Executor* and the web site under test.

Step 3: Implementing event types. Finally, the platform needs support for any relevant event types. These extensions are similar to those for the custom event type described in Sect. 5.4.2. The event types extend the `GEvent` interface. For each event type, we need to specify the classes of GUI components supporting the event and how the event is actually performed in the supported components. Table 10 shows the summary of the event types implemented for our two example platforms. As we expect, some event types (e.g., `submit`) are platform-specific.

GUITAR has been extended with Executor implementations for several common GUI platforms. Table 11 shows all platforms currently supported by GUITAR and the underlying native GUI automation library used. The human effort required for these platform-specific extensions varied considerably. For example, iOS, UNO, and Web implementations took considerably longer than their Java counterparts (typically one month by 4-member teams of undergraduate software engineering students). We attribute this difference primarily to the extra implementation required to interface between the Java core of GUITAR and the platform's native implementation.

In summary, we observe the following about GUITAR through our experiences with Case Study 5:

Table 11 GUI platforms supported by GUITAR

GUI platform	Native GUI automation library
Java JFC	Java Accessibility Framework
Web	Selenium Web Driver
Java SWT	Java SWT Accessibility Framework ^a
Android	Robotium Framework
iOS	iOS Simulator ^b
UNO (Open Office)	UNO Accessibility Framework ^c

^a<http://wiki.eclipse.org/Accessibility>

^b<http://developer.apple.com>

^c<http://openoffice.org/ui/accessibility>

1. GUITAR supports high levels of code reuse across different platforms.
2. Architecturally, GUITAR isolates platform extensions from core tool algorithms to minimize dependencies.
3. Platform extension depends on available GUI automation libraries.
4. GUITAR currently supports six different GUI platforms, demonstrating its flexibility.

6 Concluding remarks

Given the six case studies presented, we now conclude with a consideration of conclusions about the GUITAR framework based on the observations from each case study. These conclusions motivate our future work.

Considering the observations of the six case studies, we find four positive observations about GUITAR:

1. GUITAR provides a framework for model-based testing based on—(a) sampling from graph models ensuring quantified coverage (Case Study 0), (b) testing from implementation rather than specification (Case Study 0) and (c) integration into larger testing workflows (Case Study 2).
2. In some cases, e.g., when terminal events and custom GUI components are similar, manual configuration effort from the *Ripper* carries over across applications (Case Study 0, Case Study 4).
3. GUITAR's tools and algorithms (test case generation, data collection during test execution, custom GUI components, custom GUI events) can be easily extended and customized, requiring the extension of a single abstract class (Case Studies 1, 3, 4, and 4, respectively).
4. With the support of third-party, platform-specific GUI automation libraries, developers can extend GUITAR to support testing on alternative GUI platforms. Adding new platforms, while tedious, supports development of all tools by extension of a small set of abstract classes. We currently run GUITAR on six different GUI platforms, with some being more mature than others (Case Study 5).

From these, we conclude that GUITAR provides flexible support for automation, including support for customization, without sacrificing code and algorithm reuse.

GUITAR has certain limitations based on results from the case studies, all related to the *Ripper*. These “open issues” in GUITAR can be summarized as:

1. The GUI Tree generated by the *Ripper* requires manual validation (possibly assisted with XML parsing scripts).
2. Window and component identification issues during ripping can lead to inaccurate GUI Trees.
3. The *Ripper* algorithm's execution within a single application instance also leads to the generation of inaccurate GUI Trees, and particularly, false positives when using automated fault detection.

These limitations of GUITAR (and primarily the *Ripper*) inspire future work. Issue 1 is being addressed by developing a new tool which combines a visualization of the GUI Tree with a visualizations of the application's screenshots captured by the *Ripper*. This new tool will help in visual identification of inconsistencies between the application's GUI and the GUI Tree. Custom Ripper Adapters (Sect. 3.1.1) make the acquisition of fine-grained screenshots possible.

Issue 2 arises because the *Ripper* identifies GUI components based on properties such as title and icon. For some applications, these properties can change at runtime within the same GUI component. For example, a window title may change from "Document1" to "* Document1"; a "play" icon may change to a "stop" icon. This well-known problem is often referred to as the widget identification problem in GUI testing (Ruiz and Price 2008; McMaster and Memon 2009).

One solution to solve this problem is to use regular expressions when identifying widgets based on their properties (especially window titles). We are also exploring integration of computer vision-based methods for window and component identification, such as the techniques used in Sikuli Test (Chang et al. 2010).

Issue 3 arises because the *Ripper* operates entirely within a single application instance. Executing events in every possible context is considered intractable for non-trivial GUIs, although this approach would generate a very accurate GUI Tree. An alternative being considered is to combine GUI Trees generated from two more *Ripper* executions which are controlled using independent configurations.

We are also developing alternative workflows that are less sensitive to the accuracy of GUI models. In particular, the results of test case execution (even if execution was unsuccessful and would lead to a false positive, as seen in Sect. 3) can be used to update the GUI Tree, EFG, and test cases. The feedback can iteratively improve the accuracy of the model. Another possible solution is to use machine learning algorithms to repair infeasible test cases (Huang et al. 2010).

Finally, a capture tool is being developed for GUITAR. While capture tools are not natively model-based, they provide a direct mapping between the GUI model and test case execution. This tool will build on GUITAR's platform-specific models of GUI components and events so that captured test cases can be integrated directly into GUITAR's existing model-based workflow.

Another area of future work is extensive comparison to other testing frameworks, both model-based and otherwise. Research ideas such as new models (Brooks and Memon 2007), new coverage criteria (Yuan et al. 2011), new test case generation methods (Yuan and Memon 2010), and new testing workflows (Memon and Xie 2005) have been the primary motivation of GUITAR's development. While this motivation remains, we recognize that GUITAR as a tool has reached a maturity level capable of supporting more practical use cases. As our future work has outlined, we are currently

focused on adding features to GUITAR that will allow common tools (e.g., capture tools) to leverage model-based advantages. We recognize that adding features alone does not serve to legitimize GUITAR as a framework. Future studies should compare GUITAR to industry-standard frameworks, focusing on aspects of the framework that add real value for quality assurance professionals, such as GUITAR's effectiveness at finding faults, efficiency in executing feasible numbers of test cases with a reasonable turnaround, and scalability in being applied to real-world applications.

References

- Alfaro, L.D., Henzinger, T.A.: Interface theories for component-based design. In: Proceedings of the First International Workshop on Embedded Software (EMSOFT '01), pp. 148–165. Springer, London (2001)
- Amalfitano, D., Fasolino, A.R., Tramontana, P.: A GUI crawling-based technique for Android mobile application testing. In: Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11), pp. 252–261. IEEE Comput. Soc., Los Alamitos (2011)
- Artzi, S., Dolby, J., Jensen, S.H., Møller, A., Tip, F.: A framework for automated testing of javascript web applications. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), pp. 571–580. ACM, New York (2011)
- Baresi, L., Young, M.: Test oracles. Technical report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, USA (2001)
- Belli, F.: Finite-state testing and analysis of graphical user interfaces. In: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSR '01), p. 34. IEEE Comput. Soc., Washington (2001)
- Brooks, P.A., Memon, A.M.: Automated GUI testing guided by usage profiles. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE '07), pp. 333–342. ACM, New York (2007)
- Brooks, P., Robinson, B., Memon, A.M.: An initial characterization of industrial graphical user interface systems. In: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation (ICST 2009). IEEE Computer Society, Washington (2009)
- Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08), pp. 209–224. USENIX Association, Berkeley (2008)
- Chang, T.H., Yeh, T., Miller, R.C.: GUI testing using computer vision. In: Conference on Human Factors in Computing Systems, pp. 1535–1544 (2010)
- Chen, J., Subramaniam, S.: A GUI environment to manipulate FSMs for testing GUI-based applications in Java. In: Conference on System Sciences, vol. 9, p. 9061 (2001)
- Chen, W.K., Tsai, T.H., Chao, H.H.: Integration of specification-based and CR-based approaches for GUI testing. In: Conference on Advanced Information Networking and Applications, pp. 967–972 (2005)
- Cunha, M., Paiva, A., Ferreira, H., Abreu, R.: PETTool: a pattern-based GUI testing tool. In: International Conference on Software Technology and Engineering, pp. 202–206 (2010)
- Draheim, D., Lutteroth, C., Weber, G.: A source code independent reverse engineering tool for dynamic web sites. In: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR '05), pp. 168–177. IEEE Comput. Soc., Washington (2005)
- Ganov, S., Kilmar, C., Khurshid, S., Perry, D.: Test generation for graphical user interfaces based on symbolic execution. In: Proceedings of the International Workshop on Automation of Software Test (2008)
- Grechanik, M., Xie, Q., Fu, C.: Creating GUI testing tools using accessibility technologies. In: Conference on Software Testing, Verification, and Validation, pp. 243–250 (2009)
- Hellmann, T.D., Hosseini Khayat, A., Maurer, F.: Supporting test-driven development of graphical user interfaces using agile interaction design. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, pp. 444–447 (2010)
- Huang, S., Cohen, M.B., Memon, A.M.: Repairing GUI test suites using a genetic algorithm. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10), pp. 245–254. IEEE Comput. Soc., Washington (2010)

- Jääskeläinen, A., Katara, M., Kervinen, A., Maunumaa, M., Pääkkönen, T., Takala, T., Virtanen, H.: Automatic GUI test generation for smartphone applications—an evaluation. In: Proceedings of the Software Engineering in Practice Track of the 31st International Conference on Software Engineering (ICSE 2009), pp. 112–122 (companion volume). IEEE Computer Society, Los Alamitos (2009)
- McMaster, S., Memon, A.M.: An extensible heuristic-based framework for GUI test case maintenance. In: Proceedings of the First International Workshop on TESTING Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS '09). IEEE Computer Society, Washington (2009)
- Memon, A.M.: An event-flow model of GUI-based applications for testing. *Softw. Test. Verif. Reliab.* **17**, 137–157 (2007)
- Memon, A.M., Nguyen, B.N.: Advances in automated model-based system testing of software applications with a GUI front-end. In: Zelkowitz, M.V. (ed.) *Advances in Computers*, vol. 80, pp. 121–162. Academic Press, San Diego (2010)
- Memon, A.M., Xie, Q.: Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.* **31**(10), 884–896 (2005)
- Memon, A.M., Banerjee, I., Nagarajan, A.: GUI ripping: reverse engineering of graphical user interfaces for testing. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03), p. 260. IEEE Computer Society, Washington (2003)
- Memon, A.M., Pollack, M.E., Soffa, M.L.: Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng.* **27**(2), 144–155 (2001a)
- Memon, A.M., Soffa, M.L., Pollack, M.E.: 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, vol. 26, pp. 256–267. ACM, New York (2001b)
- Mesbah, A., van Deursen, A.: Invariant-based automatic testing of AJAX user interfaces. In: Proceedings of the 31st International Conference on Software Engineering (ICSE '09), pp. 210–220. IEEE Computer Society, Washington (2009)
- Myers, B.A.: User interface software tools. *ACM Trans. Comput.-Hum. Interact.* **2**(1), 64–103 (1995)
- Nguyen, D.H., Strooper, P., Suess, J.G.: Model-based testing of multiple GUI variants using the GUI test generator. In: Workshop on Automation of Software Test, pp. 24–30 (2010)
- Paiva, A.C.R., Faria, J.C.P., Mendes, P.M.C.: Reverse engineered formal models for GUI testing. In: Proc. of Conf. on Formal Methods for Industrial Critical Systems, 4916(1), pp. 218–233 (2008)
- Robinson, B., Francis, P., Ekdahl, F.: A defect-driven process for software quality improvement. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08), pp. 333–335. ACM, New York (2008). doi:[10.1145/1414004.1414072](https://doi.org/10.1145/1414004.1414072)
- Ruiz, A., Price, Y.W.: GUI testing made easy. In: Proceedings of the Testing: Academic & Industrial Conference—Practice and Research Techniques, pp. 99–103. IEEE Computer Society, Washington (2008)
- Shneiderman, B., Plaisant, C., Cohen, M., Jacobs, S.: *Designing the User Interface: Strategies for Effective Human–Computer Interaction*, 5th edn. Addison-Wesley, Reading (2009)
- Silva, J.L., Campos, J.C., Paiva, A.C.R.: Model-based user interface testing with spec explorer and ConcurTaskTrees. *Electron. Notes Theor. Comput. Sci.* **208**, 77–93 (2008)
- Silva, J.C., Saraiva, J., Campos, J.C.: A generic library for GUI reasoning and testing. In: ACM Symposium on Applied Computing, pp. 121–128 (2009)
- Staiger, S.: Static analysis of programs with graphical user interface. In: 11th European Conference on Software Maintenance and Reengineering (CSMR '07), pp. 252–264 (2007)
- Ural, H., Yang, B.: A test sequence selection method for protocol testing. *IEEE Trans. Commun.* **39**(4), 514–523 (1991)
- Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with Spec Explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) *Formal Methods and Testing*, Chap.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, pp. 39–76. Springer, Berlin (2008)
- Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., Kazmeier, J.: Automation of GUI testing using a model-driven approach. In: Conference on Software Engineering, pp. 9–14 (2006)
- Xie, Q., Memon, A.M.M.: Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.* **16**(1), 4 (2007)
- Xie, Q., Memon, A.M.: Using a pilot study to derive a GUI model for automated testing. *ACM Trans. on Softw. Eng. and Method.* (2008)

- Yuan, X., Memon, A.M.: Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Trans. Softw. Eng.* **36**, 81–95 (2010)
- Yuan, X., Cohen, M.B., Memon, A.M.: GUI interaction testing: incorporating event context. *IEEE Trans. Softw. Eng.* **37**(4), 559–574 (2011)
- Zhang, S., Saff, D., Bu, Y., Ernst, M.D.: Combined static and dynamic automated test generation. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*, pp. 353–363. ACM, New York (2011)