JPF-AWT: Model Checking GUI Applications

Peter Mehlitz, Oksana Tkachuk
NASA Ames Research Center
Moffett Field, CA, USA
peter.c.mehlitz,oksana.tkachuk@nasa.gov

Mateusz Ujma*
Department of Computer Science
University of Oxford, UK
mateusz.ujma@cs.ox.ac.uk

***Abstract*—Verification of Graphical User Interface (GUI) applications presents many challenges. GUI applications are open systems that are driven by user events. Verification of such applications by means of model checking therefore requires a user model in order to close the state space.**

In addition, GUIs rely extensively on complex and inherently concurrent framework libraries such as AWT/Swing, for which the application code merely provides callbacks. Software model checking of GUI applications therefore needs abstractions of such frameworks that faithfully preserve application behavior.

This paper presents JPF-AWT, an extension of the Java PathFinder software model checker, which addresses these challenges. JPF-AWT has been successfully applied to a GUI front end of a NASA ground data system.

I. Introduction

A Graphical User Interface (GUI) is a suitable mechanism to hide the underlying application's complexity from the user. GUIs simplify a user's task by guiding through a series of windows, enabling or disabling relevant components and validating user input before it is processed. While a GUI can significantly improve the usability of an application, it can also impose severe challenges for the development and verification of such applications.

It is important to verify GUI behavior. Examples of GUI properties may include response requirements (e.g., clicking on button A opens window B), or checking that the user can always make progress (i.e., no "user shutout"). Verification of GUIs is challenged by a number of GUI-specific aspects:

(1) GUI applications are *open event-driven* systems that engage in long running interactions with users and other software or hardware components. The number of possible interaction sequences may be too large for manual or even automated testing.

(2) GUI applications are typically implemented on top of *large frameworks* such as Java's Abstract Window Toolkit (AWT) [10] or Swing [11] libraries. The application code mostly provides callbacks for the framework, which is responsible for obtaining user input and dispatching it to the respective user interface components such as buttons and lists. A large part of the verification relevant behavior of the system under test is defined by the framework, not the application code itself.

(3) GUI applications are inherently *concurrent*. Even if the application itself is not multi-threaded, the framework always uses multiple processes or threads (e.g., event dispatcher thread, X server process). On top of the framework concurrency, GUI applications have to employ explicit concurrency for lengthy computations in order to guarantee responsiveness of the user interface (e.g., the "0.1 sec rule"). In addition, large parts of the GUI framework are system global, i.e., shared between different applications that should not affect each other.

Current approaches to GUI validation include unit testing (e.g., [4], [5], [12]), capture-replay (e.g., [6], [9], [13]), and model-based testing (e.g., [7], [8], [14]). Unit testing techniques usually test short sequences of user events, largely created manually. Capture-replay approaches require a user to record interaction sequences with the GUI and then automatically replay them during testing. Both approaches require test sequences to be described explicitly, therefore, requiring large resources to create extensive test suites.

Model-based approaches are capable of producing large sets of test sequences based on a model specification. For example, Memon et al. [7], [8], [14] automatically extract directed graph models, while dynamically executing the GUI, and use the extracted graphs to generate test sequences. However, runtime model extraction and testing cannot guarantee full coverage of the program behavior, especially in the presence of concurrency.

Model checking techniques are specifically designed to exercise all possible thread interleavings of the system. However, in the context of large and complex GUI frameworks, software model checking has a huge barrier to cross in terms of scalability. Previous work by Dwyer et al. [2] attempted to make model checking tractable by aggressively stubbing out AWT/Swing libraries, as a result producing GUI models with only one event dispatching thread. This single thread assumption may not be appropriate for real applications.

In this paper, we present a tool for model checking GUI applications that addresses the challenges of model checking GUIs by (1) allowing the user to specify sets of interaction sequences using a simple yet powerful formal notation, (2) incorporating support for the actual AWT/Swing libraries, and (3) employing the model checking engine of Java PathFinder (JPF) [1], [3] to verify all possible thread interleavings and input sequences. The tool is called JPF-AWT and is implemented as a JPF extension.

The remainder of the paper is organized as follows: Section II presents a motivating example, Section III provides

* At the time when this research was conducted, Mateusz Ujma was an intern at NASA Ames Research Center.

an overview of the tool design, Section IV discusses the experience gained while successfully applying the tool to a NASA ground data system, and Section V concludes.

II. Example

To demonstrate the challenges of GUI model checking, we will use a `RobotManager` example application that controls a number of robots. Figure 1 shows the GUI of this `RobotManager` application.

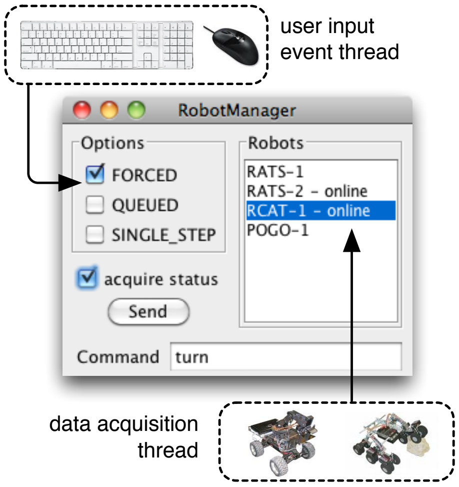


Fig. 1. Robot Manager GUI

The program is used to control four robots, as shown in the `Robots` list: `RATS-1`, `RATS-2`, `RCAT-1` and `POGO-1`. Each robot can be online or offline, as indicated within the respective list entry. The user can enter commands in a text field; commands can be further attributed by selecting any of the three `FORCED`, `QUEUED` and `SINGLE_STEP` checkboxes.

The nominal control procedure consists of the following steps:

1) entering a command
2) selecting command options (if any)
3) selecting a robot from the list
4) sending the command to the robot by means of clicking the `Send` button.

By checking the `acquire status` checkbox, the user can start a background thread that probes robots for their online status, and dynamically updates the related entries of the `Robots` list as each robot becomes online or offline.

To verify the behavior of the `RobotManager` example, one needs to test not only all possible combinations and permutations of commands and options, but also all possible thread interleavings between the event dispatcher thread of the framework and the data acquisition thread of the application. Especially the second aspect is usually beyond the reach of traditional testing techniques.

III. Tool Description

JPF-AWT is implemented as an extension of JPF [3], which is an open source, explicit state software model checker for Java bytecode. JPF features its own Java Virtual Machine (JVM) that is capable of storing, matching and restoring the state of the executed bytecode program. Off the shelf, JPF can check for property violations such as deadlocks, data races and unhandled exceptions, using on-the-fly partial order reduction and other techniques to reduce the number of program states that have to be explored. However, JPF's main quality is to provide a number of well defined extension mechanisms that allow

- introduction of new state space branches other than scheduling points (extensible *ChoiceGenerators* [3])
- observation of state space exploration events such as backtracking (*Listeners* [3])
- abstraction of standard libraries (*Model* and *NativePeer* classes [3])

Our JPF-AWT tool makes use of all three extension mechanisms.

Since JPF verifies Java bytecode, JPF-AWT targets GUI applications that use the standard Java AWT and Swing framework libraries.

A. Architecture

The main objective for the JPF-AWT design is to check the application behavior against a potentially large set of different input sequences, while preserving as much of the GUI framework as possible, which includes threading structure, window composition and callback notification. This is achieved by replacing only the low-level, platform-specific parts of the framework libraries that handle rendering and input acquisition, leaving the application itself completely unmodified. Figure 2 shows the high-level architecture of JPF-AWT. Since

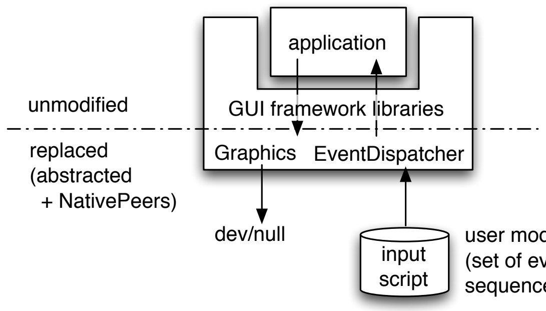


Fig. 2. JPF-AWT Architecture

most of our targeted program properties are either functional (assertions), or refer to generic defects such as unhandled exceptions and race conditions, we ignore graphical output. This is done by means of providing modeled versions of the respective library classes such as `java.awt.Graphics`, to stub out calls to native rendering code.

We also replace library classes that interface to the platform-specific windowing system (such as `java.awt.Window`), carefully preserving component composition and callback behavior.

Conceptually, the main component of JPF-AWT is a replaced `java.awt.EventDispatchThread` which models non-deterministic user input, using JPF's *NativePeer* mechanism to interface to a scripting engine.

B. User Input Modeling

Input sequences are specified by means of a simple scripting language, which uses *events* as its basic building block, consisting of a target component identifier, the name of a method within the related component class, and optional string or numeric arguments:

$$\$\langle componentId\rangle.\langle methodName\rangle(\langle arg\rangle,\ldots)$$

Component identifiers are either names that are explicitly set by the application code or labels that are uniquely associated with the respective component, such as button or text field labels.

Component identifiers, method names and argument values can use patterns for string alternatives and character sets, which are expanded into a set of respective combinations:

$$< aaa|bbb > ([1-2]) \Rightarrow aaa(1), aaa(2), bbb(1), bbb(2)$$

The language provides three special constructs: **ANY** elements represent non-deterministic input choices to be explored by the model checker, **REPEAT** elements are used to specify loops over event sequences, and **NONE** events simply advance to the next sequence event.

Figure 3 shows an example script for the `RobotManager` application described in section 2.

```
// start the data acquisition thread
$acquire_status.doClick()

// enter command into "Command" text field
$Command:input.setText("turn")

// select options checkbox combinations
REPEAT 3 {
  ANY { NONE,$<FORCED|QUEUED|SINGLE_STEP>.doClick() }
}

// select a target robot (list selection)
ANY { $Robots:list.setSelectedIndex([0-3]) }

// click on "Send" button
$Send.doClick()
```

Fig. 3. RobotManager Input Script

JPW-AWT scripts are processed by an interpreter which stores and restores its internal state by means of a JPF *listener*. While the `EventDispatchThread` is a modeled class that is executed by JPF, obtaining the next event from the script interpreter is delegated to the *NativePeer* of this class, which executes at the host VM level and can therefore directly access JPF functions. This mechanism resembles the use of *native methods* in a standard Java VM.

The reason for processing script events outside of JPF executed code is to avoid the creation of event objects that would alter the program state of the system under test, which is especially important to close the state space for unbounded **REPEAT** loops.

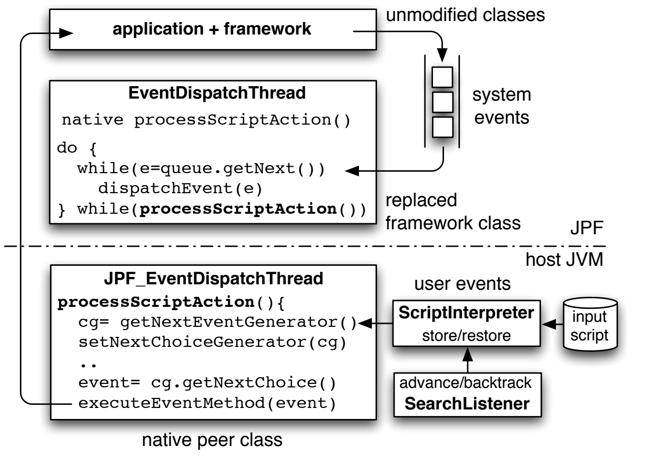


Fig. 4. JPF-AWT Input Management

Figure 4 shows the structure of the JPW-AWT input model. The respective `processScriptAction()` method within the `EventDispatchThread` native peer class obtains the next input event from the script interpreter, uses JPF's reflection mechanism to map the event to the corresponding method in the target component class, and then directs the JPF VM to execute this method. In case the script interpreter returned an **ANY** element, `processScriptAction()` causes JPF to store the current program state and repeats this process for each choice within the **ANY** event set.

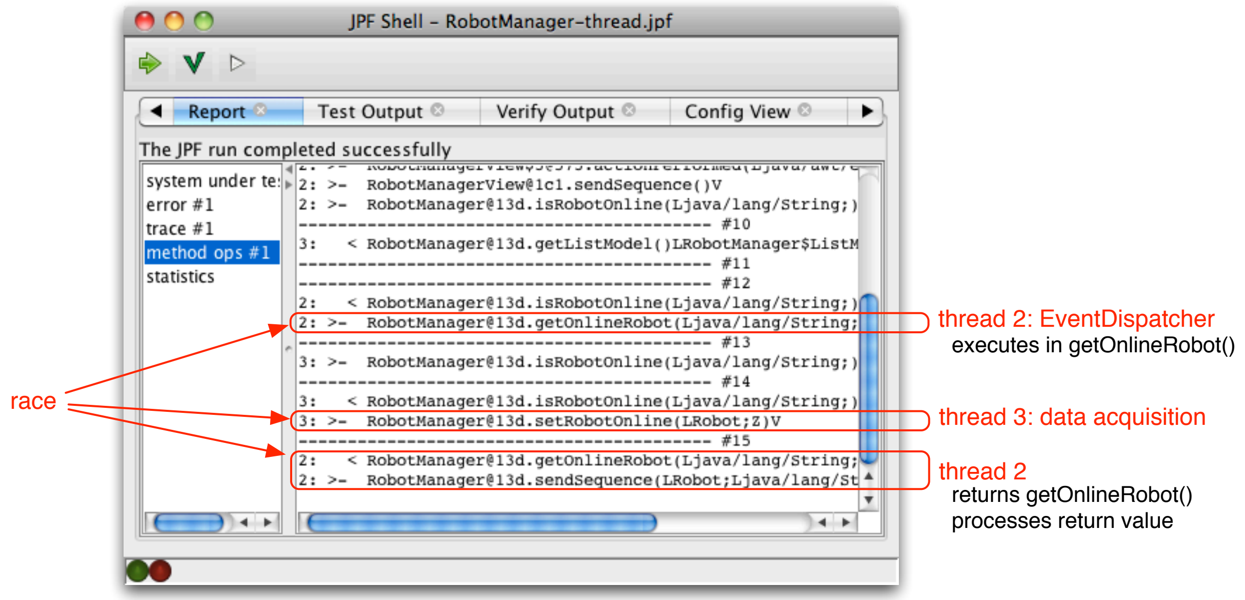
The event sequence exploration mechanism of JPF-AWT is completely orthogonal to other state space branches processed by JPF. In particular, it does not interfere with JPF's model checking for concurrency defects and is therefore well suited to detect defects that are complex interactions between certain scheduling sequences and user input combinations.

IV. Experience

Using the specification shown in Figure 3, model checking the `RobotManager` application produces the results shown in Figure 5. JPF-AWT is able to find a null pointer exception that is very hard to reproduce using traditional testing techniques. The exception is caused by a race condition between the data acquisition thread and the `EventDispatchThread`, the first one setting the selected robot offline right in the middle of the send button click processing.

Figure 6 shows a deeper analysis of this defect by means of specialized JPF listener that detects overlapping method calls on the same object from different threads. The analysis is performed in a graphical shell for JPF, which is compatible with JPF-AWT and can even be extended to show JPF-AWT specific verification reports.

In addition to verifying small examples like the `RobotManager`, we applied JPF-AWT to one of the NASA's large ground data systems. First, we translated its user models into the script language defined in Section III. Second, we defined the application properties using assertions. Assertions can be as simple as `assert errorList.size() == 0`



The JPF run completed successfully

```
2: >-  RobotManagerView@1c1.sendSequence()V
2: >-  RobotManager@13d.isRobotOnline(Ljava/lang/String;)
------------------------------------------ #10
3:   < RobotManager@13d.getListModel()LRobotManager$ListM
------------------------------------------ #11
------------------------------------------ #12
2:   < RobotManager@13d.isRobotOnline(Ljava/lang/String;)
2: >-  RobotManager@13d.getOnlineRobot(Ljava/lang/String;
------------------------------------------ #13
3: >-  RobotManager@13d.isRobotOnline(Ljava/lang/String;)
------------------------------------------ #14
3:   < RobotManager@13d.isRobotOnline(Ljava/lang/String;)
3: >-  RobotManager@13d.setRobotOnline(LRobot;Z)V
------------------------------------------ #15
2:   < RobotManager@13d.getOnlineRobot(Ljava/lang/String;
2: >-  RobotManager@13d.sendSequence(LRobot;Ljava/lang/St
```

Fig. 6. Analysis of RobotManager Defect

```
================================================= error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.NullPointerException: Calling
   'processSequence(Ljava/lang/String;)Ljava/lang/String;'
          on null object
       at RobotManager.sendSequence(RobotManager.java:265)
       ...
================================================= statistics
elapsed time: 0:00:03
states:        new=1320, visited=207, backtracked=1490, end=0
search:        maxDepth=68, constraints hit=0
choice gens:   thread=41 (signal=0, lock=10, shared ref=28),
               data=1444
heap:          new=21524, released=11537, max live=3333,
               gc-cycles=1480
instructions: 1677327
max memory:    81MB
loaded code:   classes=404, methods=4959
```

Fig. 5. Robot Manager Verification Results

or very complex, including checking for deadlocks and race conditions. It is important to mention that assertions are written in Java and are part of the verified application. As a result, they can also be used to verify the application during runtime by traditional testing techniques. In a last step, we also created JUnit [5] tests that run JPF-AWT according to the specified scripts.

Applying JPF-AWT to this project confirmed the usefulness of our approach. Despite the fact that the project already had an extensive test suite, our method found 12 previously unknown errors. We also achieved significantly better coverage, increasing the number of verified user scenarios from 370 to more than 28,000. The effort to create scripts, assertions and JUnit tests was comparable to the effort to develop the existing test suite (which did not catch the defects). The only observable downside was an increase in test time from seconds to minutes, which is in our opinion an acceptable trade-off.

V. Conclusion

We presented JPF-AWT, an extension of the Java PathFinder software model checker to efficiently verify potentially large GUI applications. Given a script specifying sequences of user inputs, JPF-AWT can directly check the unmodified, compiled application. JPF-AWT preserves the threading structure of the system under test, and can therefore find defects that are caused by complex interaction between user inputs and scheduling sequences. JPF-AWT has been successfully applied to a large NASA ground data system, finding defects that had escaped conventional testing.

Planned future work includes extensions of the input scripting language and extraction of initial scripts using capture-replay or runtime techniques (e.g., [7], [8]).

References

[1] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
[2] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser. Analyzing interaction orderings with model checking. In *ASE'04: Proceedings of the 19th IEEE international Conference on Automated Software Engineering*, pages 154–163. IEEE Computer Society, 2004.
[3] Java PathFinder. Website. http://babelfish.arc.nasa.gov/trac/jpf.
[4] jfcUnit. Website. http://jfcunit.sourceforge.net/.
[5] JUnit. Website. http://www.junit.org.
[6] Marathon. Website. http://sourceforge.net/projects/marathonman.
[7] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 260. IEEE Computer Society, 2003.
[8] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, 2005.
[9] SeleniumIDE. Website. http://seleniumhq.org/projects/ide.
[10] SUN. Abstract Windowing Toolkit. http://java.sun.com/products/jdk/awt/.
[11] SUN. Swing. http://download.oracle.com/javase/6/docs/technotes/guides/swing/.
[12] UISpec4J. Website. http://www.uispec4j.org.
[13] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using gui screenshots for search and automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM.
[14] X. Yuan, M. B. Cohen, and A. M. Memon. Gui interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37(4):559–574, 2011.