

# Analyzing Interaction Orderings with Model Checking

Matthew B. Dwyer, Robby, Oksana Tkachuk  
Kansas State University  
Manhattan, KS 66506, USA  
{dwyer,robby,oksana}@cis.ksu.edu

Willem Visser  
RIACS, NASA Ames Research Center  
Moffett Field, CA 94035, USA  
wvisser@email.arc.nasa.gov

## Abstract

*Human-Computer Interaction (HCI) systems control an ongoing interaction between end-users and computer-based systems. For software-intensive systems, a Graphic User Interface (GUI) is often employed for enhanced usability. Traditional approaches to validation of GUI aspects in HCI systems involve prototyping and live-subject testing. These approaches are limited in their ability to cover the set of possible human-computer interactions that a system may allow, since patterns of interaction may be long running and have large numbers of alternatives.*

*In this paper, we propose a static analysis that is capable of reasoning about user-interaction properties of GUI portions of HCI applications written in Java using modern GUI Frameworks, such as Swing<sup>TM</sup>. Our approach consists of partitioning an HCI application into three parts: the Swing library, the GUI implementation, i.e., code that interacts directly with Swing, and the underlying application. We develop models of each of these parts that preserve behavior relevant to interaction ordering. We describe how these models are generated and how we have customized a model checking framework to efficiently analyze their combination.*

## 1 Introduction

Modern software is becoming increasingly feature-rich and integrated into mission critical processes. A Graphical User Interface (GUI) serves to foster efficient and effective Human-Computer Interaction (HCI) by: (a) depicting application data in forms that allow humans to quickly and clearly understand that data, and (b) by guiding and controlling interaction by presenting only the set of allowable actions that users may perform next based on the underlying application's data state. Traditionally, GUI validation has involved prototyping a graphic user interface and performing live-subject testing to assess both of these concerns. Live-subject testing is clearly necessary, especially in assessing usability and ergonomic aspects of GUIs, but

the potentially long-lived nature of the interactions between users and system and the number of alternatives available to a user at each step in an interaction make coverage testing of the space of possible interaction orders infeasible.

In this paper, we present a static approach to analyzing program behavior under all possible interaction orderings enforced by a GUI system implemented in the Java Swing framework. We discuss Swing in detail in Section 3, but it is important to understand that it presents several challenges for static analysis.

**Swing is an object-oriented framework** where windows, widgets on a window (e.g., buttons, selections, text entry boxes), the text and color associated with widgets and windows, and numerous additional attributes are all defined by instantiating framework classes. References among those class instances define information about the visibility, modality, and enabledness of widgets that is crucial to defining the evolving state of the user-interface in response to user-initiated actions.

**Swing is an event-driven framework** that inverts and internalizes the application's control flow. Programmers define methods, called *event-handlers*, that respond to occurrences of user-initiated actions, called *events*, like the pressing of a button. Handler methods are bound to events by making framework calls that record the *event-handler* relation. The framework executes a cyclic *event-dispatching thread*, which processes each event in turn and invokes the associated handler-methods.

The fact that object values and object referencing relationships are used to define the structure and behavior of a Swing GUI, means that traditional static analysis approaches that do not capture program data values precisely are of limited use; a more precise, object-sensitive form of analysis is needed. This observation led us to explore the use of model checking as a means of reasoning about GUI behavior in previous work [8]. In this paper, we apply *software model checking* approaches [5, 18, 22] to automate and scale the analysis of interaction orderings in Java GUIs to realistic systems. Our approach consists of partitioning an HCI application into three parts: the Swing framework, the *GUI implementation* (i.e., the GUI setup and event-handler

code that interacts with Swing to configure the structure of the GUI and to implement its control-logic), and the *underlying application* (i.e., the code that is common to GUI and command-line versions of an application). By decomposing an application in this way we can target each part with a different technology for extracting a faithful and appropriately precise model of its behavior. This is crucial since it is well-known that the cost of model checking can grow exponentially with the size of the system under analysis. To enable efficient model checking of GUI interaction ordering-related behavior, we have developed techniques for generating GUI models with a single event-dispatching thread, for efficiently exploring the large space of options available to the user at each step in the interaction, and for abstracting the data state of the underlying application.

We use the Bandera Environment Generator (BEG) [20, 21] to perform precise inter-procedural side-effects analysis of the Swing framework to understand which framework classes and methods can influence event-handler execution; the results of this analysis form an initial approximate behavioral model of each framework method that is subsequently refined manually. We also use BEG to automatically generate safely approximating models of the underlying application’s classes and methods. These application models sacrifice precision for scalability by collapsing application data to a small number of representative abstract values. As discussed in Section 3, this abstraction does not impede analysis since GUI implementations are not typically sensitive to specific application data values. Finally, we use model extraction techniques from Bandera [5] to generate finite-state models that capture the control-flow constructs, local computation, and Swing and application method calls of the GUI implementation. The resulting combination of models safely captures event-related GUI behavior while abstracting other aspects of the GUI, e.g., color, shape, size, and the underlying application. Furthermore, the model retains the structure of an event-driven Swing system. We exploit that structure to minimize the number of visible program states that are stored during analysis for GUIs with a single event-dispatching thread. The implementation of this optimization in our customizable model checking framework, Bogor [18], required only minor modification, yet, as we show in Section 5, it can significantly reduce the cost of analysis.

This paper makes several contributions, including (i) presentation of a compositional approach to model construction that exploits the structure of GUI applications, (ii) description of a semi-automated method for generating models of complex library frameworks, (iii) description of approaches for customizing model checking algorithms to exploit the semantics of GUI frameworks for state-space reduction, and (iv) preliminary evidence that the overall approach scales to treat features found in real Java Swing GUI code. Finally, we believe that this work provides further

evidence of the need for flexible and customizable model checking frameworks, such as Bogor, to target and exploit properties of specific software domains in order to achieve efficient and precise analysis.

The next Section gives an overview of our analysis approach and gives some background on our model checking tools. Section 3 describes the Swing framework and our model of framework behavior as it relates to the ordering of user-interactions and the execution of event-handler code. Section 4 gives an overview of the Bogor model checking framework and discusses several optimizations that have been developed to reduce the cost of analysis. We describe the application of our method and tools to several Java GUIs in Section 5 and assess the potential for scaling to larger applications. Section 6 discusses related work and we conclude in Section 7.

## 2 Background and Overview

The field of software model checking has developed rapidly over the past five years. Tools that can analyze Java programs of up to ten-thousand lines of code with tens of threads are now available [22]. The fundamental complexity of this form of analysis, however, is inescapable. Progress has been made by developing increasingly sophisticated techniques for identifying semantics preserving state-space abstraction and reduction techniques that allow for certain details of a systems behavior to be skipped during analysis. In this section, we briefly describe two frameworks for implementing forms of abstraction and reduction for Java programs that form the basis of our analyses of Swing GUI implementations.

**Bandera Environment Generator (BEG)** [20, 21] is a framework for analyzing specifications and Java implementations and synthesizing safely approximating models of their behavior. The most relevant aspects of BEG for this paper are its alias and side-effects analyses. BEG analyzes the influence of a part of a code base on specified program classes or fields. One designates a collection of relevant classes or fields, called the *unit*, and a collection of methods (or classes) to be analyzed. BEG calculates a summary of the side-effects that each method may make on the unit; note that the summary encodes the transitive effects of the entire call-tree descending from the named method.

BEG uses a context-sensitive access-path based alias analysis to drive the side-effects calculations and is able to refine its *may* side-effects summaries using *must* and *path-sensitive* side-effects information. These summaries are reified as Java code using special modeling primitives that encode non-deterministic choice operations over portions of the heap as a mechanism for representing sets of objects. Thus, BEG can be viewed as calculating an abstraction of a code base’s behavior relative to the unit. In previous work, we used BEG in conjunction with the Java

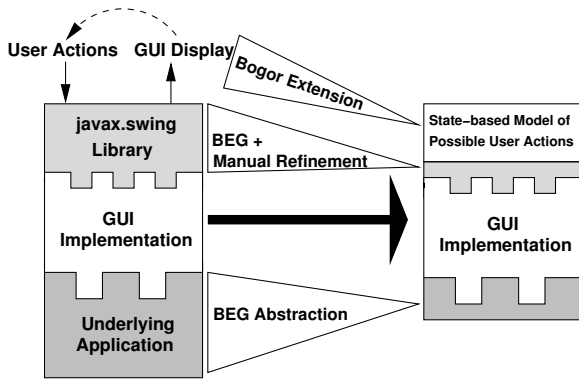


Figure 1. Model Extraction Strategy

PathFinder (JPF) model checker [22] to analyze several large applications including a multi-thousand line aircraft cockpit display simulation. For more information on BEG see <http://beg.projects.cis.ksu.edu>.

**Bogor** [18] is an extensible and highly modular software model checking framework designed to ease the development of robust and efficient domain-specific model checkers for verification of dynamic and concurrent software. In previous work, we showed that the cost of model checking can be reduced significantly (i.e., by order of magnitudes) by leveraging domain-specific information, e.g., domain-specific abstractions [6], state-space structures [10], and object access and locking disciplines [9].

In contrast to other model checkers such as Spin [13], Bogor’s modeling language (BIR) provides high-level features commonly found in modern programming languages such as dynamic creation of objects and threads, dynamic dispatch of methods, exception handling, etc. Furthermore, BIR is extensible – it allows the introduction of new abstract data types and abstract operations as first-class constructs of the language. This is analogous to adding new native types and native instructions to a core BIR virtual machine.

Another notable feature of Bogor is its modular architecture. It is designed with clean Application Programming Interfaces (API) using well-known design patterns [11]. This eases the incorporation of algorithms targeted to reduce the cost of model checking. For more information on Bogor see <http://bogor.projects.cis.ksu.edu>.

## 2.1 Our Approach

The left-side of Figure 1 illustrates the three-layer structure of a Swing application. The framework owns the main execution thread, controls the rendering of display images, and processes user inputs to produce events that are relevant to the application. The *GUI implementation* is comprised of all of the application code that *directly* manipulates Swing types, for example, to configure the structure of the GUI and to define and register event-handler methods. The *un-*

*derlying application* is the remainder of the application that is, by definition, not directly dependent on Swing types.

Our goal is to be able to reason about sequences of user interactions that a GUI implementation allows; we call these *interaction orderings*. Intuitively, the different layers of a Swing application exert different degrees of influence on the behavior of the overall program relative to interaction orderings.

As discussed in the next Section, Swing allows the definition of window objects that control the visibility and enabledness of widgets through which a user may initiate interaction. Swing’s event-handler invocation mechanism also plays an important role in defining the top-level control-flow among handler methods, which can in turn influence the set of interactions that are presented to the user. On the other hand Swing types provide an enormous variety of options for controlling the visual aspects of the GUI; these details are not relevant for reasoning about interaction orderings. Our approach is to develop a single model of the Swing framework’s interaction ordering-related behavior. The fidelity of such a model with respect to the framework’s implementation is critical if our analysis is to be useful. Towards this end, we configure BEG to preserve information about Swing’s interaction ordering-related types and fields. This produces a summary of side-effects on interaction ordering-related framework data for each method in the Swing API. This sound, but approximate, model is used as a starting point for the manual development of a more precise model. This model is reused across multiple analyses, hence the cost of its construction can be amortized.

We use techniques developed in Bandera [5] to translate a GUI implementation’s code directly into BIR, the input language of Bogor. One concern in applying Bandera’s translation is the potential for unbounded object creation, but this does not happen in the GUI implementations we have encountered. Large numbers of Swing objects are often created to define the structure of GUI, but these objects are largely invariant subsequent to initialization. Event-handler methods may create local data for performing calculations, but most data does not escape those methods and is garbage collected upon method return, and the data that does escape, for example strings written to the GUI display, are abstracted since we only preserve interaction ordering and not the exact values involved in interactions.

BEG is applied to the underlying application to calculate its effects on GUI implementation data. In our experience, these effects are minimal and the result is an extremely coarse abstraction of the application. This tends not to cause problems since the event-handler’s behavior is determined by explicitly testing return values from application method calls and the control-flow of event-handlers is modeled precisely.

Finally, the user is modeled as being able to select any input action that is visible and enabled in a given state of

the GUI. The models described above define that state and a Bogor extension, described in Section 4, is used to perform a non-deterministic choice from those input actions.

**Interaction Ordering Properties** In this paper, we focus on the abstraction and analysis of GUI interaction ordering properties and not on the properties themselves. In previous work, we explored a range of safety properties related to human-computer interactions in GUI systems [8]. In that work, we used computation tree logic [15] to express the properties, but they could just as well be expressed using pattern-based [7] regular expressions. For example, a property such as “When an error message is displayed, the only available user action is acknowledgement via the ‘ok’ button” could be checked by analyzing the system to ensure that no executions match the regular expression

```
.; display[error]; ^button[ok]; .*
```

which encodes a violation of the property (i.e., that the display of an error is followed by an action other than the press of the ok button). Bogor has an extension for checking safety properties expressed as finite-state automata so this is a natural fit in our analysis framework.

### 3 Modeling Event-Handling in Swing™

A typical GUI application creates a number of windows and widgets. As the user interacts with the application, the number of windows and widgets available for interaction changes. Figure 2 shows a simple Swing example. At the state captured on the figure, the application displays four windows: the main frame (left), the non-modal dialog (upper right) and two modal dialogs (lower right).

#### 3.1 Modeling Components

The essential step in modeling GUI components is identifying their abstract state. Swing is a very large library and rather than exhaustively describe the abstract state of each component, we describe the rationale for abstracting the components of our example.

A *modal* dialog is one that restricts the next user interaction to the enabled actions on that dialog; all other actions are disabled. We model such restriction by keeping track of modality as part of abstract state of dialogs. We keep track of all available windows in the system in two data structures: a set of windows that do not restrict user interactions (i.e., frames and non-modal dialogs) and a stack for restrictive windows (i.e., modal dialogs). At each step, if the second structure is not empty, the modal dialog on the top of the stack represents the window a user may interact with. If there are no modal dialogs open, then the user may interact with any window from the first set. In Figure 2, the set

of non-restrictive windows contains the main frame and the non-modal dialog; the stack of modal dialogs contains the other two windows, with the error message dialog on top. In this state, the user is restricted to using the error dialog.

Once the user chooses a window to work with, he or she is then able to select from that window’s components that are both *visible* and *enabled*. Invisible components and their children are not drawn on the screen, for example, the components of the left tab of the main frame of Figure 2. Disabled components do not allow for user selection. We model these aspects of components by including per widget visibility and enabledness booleans. Thus, at each step, the user may choose among enabled components that are reachable from the top-level window, following a path that consists of visible components. To keep track of parent-child relationships, we preserve containment properties, e.g., that the main frame contains a group of two tab panels, label, and buttons at the top in Figure 2. The group of tabs (or radio buttons), in addition to containment, shows selection: only one item at a time may be selected. Selecting a new component deselects the previously selected one. This is called a single selection model and we encode that by consulting containment and selection information when updating component states for radio buttons.

In addition to components that the user can click on, GUIs present components where the user may enter input data, e.g., the Quiz dialog contains an input field where the user may enter text. The actual value the user inputs on such components is not explicitly stored as part of the abstract state; operations that would operate on such *missing values* must assume that it could take on any legal value of its type.

The abstract state of the right, or More Dialogs, tab in Figure 2 is defined by the following values: *parent* : tabs selection group, *visible*, *enabled*, *selected*, and *contains* : {radio button group, Show It! button}.

#### 3.2 Modeling Event-Handling

The abstract state of each GUI component is defined by values of the fields we model; the abstract state of the entire system is defined by the state of each constituent component. When a user performs an action on a GUI component, an event of corresponding type is fired and sent to listeners subscribed to be notified of that event. The event-handling mechanism in Swing applications is an example of publish-subscribe pattern. Our model of the event-dispatch mechanism is integrated with the model of the user described in the previous section; Figure 3 shows the Java code of the model. A window is selected for interaction by prioritizing modal dialogs and choosing any top-level window, or reachable sub-window, if none exist. That window is analyzed to determine the visible and enabled components it contains and one of those is selected. The registered handlers for that

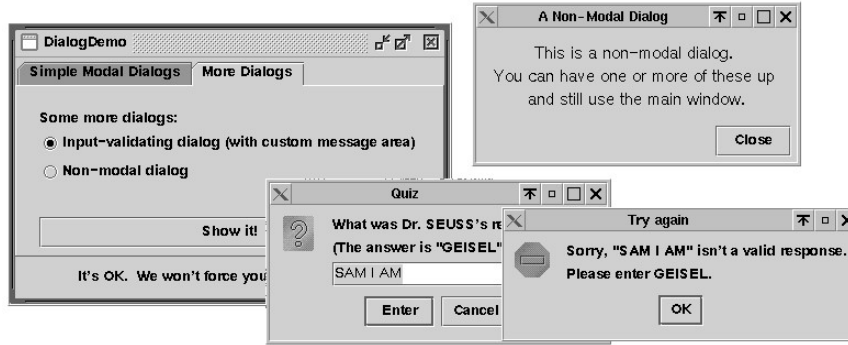


Figure 2. GUI Example with Modal and non-Modal Dialogs

```

public static void main(String[] args) {
    JComponent container; ...
    while (true) {
        window = chooseTopWindow();
        container = (JComponent) chooseReachable(
            "javax.swing.JComponent",
            window, isVisible, isEnabled);
        notifyListeners(container);
    }
}

public static Window chooseTopWindow() {
    Window window = null;
    Stack modalDialogs = Model.getRestrictiveWindows();
    if (!modalDialogs.empty())
        window = (Window) modalDialogs.pop();
    else {
        Set topWindows = Model.getNonRestrictiveWindows();
        window = (Window) chooseReachable(
            "java.awt.Window", topWindows);
    }
    return window;
}

```

Figure 3. Event-Dispatch Model (excerpts)

component are then notified in turn. The key to this model is the ability to express non-deterministic choice over collections of heap allocated objects; modeling primitives for these operations are discussed in Section 4.

The GUI events can be divided into two categories: *logical* events, which correspond to user actions that require interaction with the underlying application or GUI control-logic, and *low-level* events, which indicate actions that are primarily handled automatically by the default GUI framework, e.g., listeners that highlight a component or display a tooltip when a mouse is moved over it. We focus on logical events and their handlers in this work, although the low-level events can be treated using exactly the same mechanisms.

### 3.3 Analysis-Guided Model Definition

In Java, an object can be described by the values of its fields. For example, an instance of `java.awt.Component` defines values for over eighty fields of various types (e.g.,

Color background, int width). Recording the actual state of a Component instance by storing the values of all of its fields is very expensive. Fortunately, for verification of interaction ordering behavior, we only need to record the values of fields that relate to the logical state of the component and not its *look and feel*. In the rest of this section, we describe the process by which Swing framework methods are analyzed and how we convert those analysis results into models that can be submitted to Bogor. We focus on the Component hierarchy to make the discussion concrete.

All Swing components inherit their properties from Component, which declares boolean fields `visible` and `enabled`. `java.awt.Container` is a sub-type of Component, which implements containment properties through the field `Component[] component`. Modality is implemented by a boolean field `modal` of `java.awt.Dialog`. `javax.swing.JComponent`, a descendant of Container, declares `EventListenerList listenerList`, where listeners of `*Listener` type (e.g., `MouseListener`, `ComponentListener`) may register using `add*Listener` method. All Swing components inherit this listener mechanism.

In addition to inherited features, Swing components declare fields reflecting their specific features (e.g., tabs are implemented by `JTabbedPane`, which declares `Vector pages` to keep track of added tabs and `SingleSelectionModel` to keep track of tab selection). Integer type fields can affect the number of widgets created for a component (e.g., `int optionType` in `JOptionPane` defines how many buttons are displayed on the pane). Fortunately, such fields have a small set of predefined values (e.g., `optionType = YES_NO_OPTION` produces a pane with two buttons: yes and no).

We use side-effects analysis in BEG to guide the modeling of Swing components. Since we are interested in preserving specific properties of Swing components (e.g., containment, registration of listeners), we define a specialized side-effects analysis which calculates side-effects on specific fields of Swing components, e.g., boolean fields (`vis-`

```

public class Container extends Component{
  Component[] component = new Component[0];
  public Component add(Component comp) {
    addImpl(comp, null, -1);
    return comp;
  }
  protected void addImpl(Component comp,
                          Object constraints, int index){
    if (ncomponents == component.length) {
      Component newcomponents[]=new Component[..];
      component = newcomponents; ...
    }
    if (index == -1 || index == ncomponents) {
      component[ncomponents++] = comp;
    } else {
      component[index] = comp; ncomponents++;
    } ...
  }
}

```

```

// must side-effects
this.component[TOP_INT] = param0;
this.ncomponents = TOP_INT;
// may side-effects
Component[] component0 = new Component[TOP_INT];
...
this.component = component0;
//return locations
{ param0 }
...
public class Container extends Component {
  Component[] component;
  int ncomponents;
  public Component add(Component param0){
    component[ncomponents] = param0;
    ncomponents++;
    return param0;
  }
}

```

Figure 4. Example Swing Method, Analysis Results, and Model

ible, enabled, etc.), fields that serve as containers (arrays, lists, etc.) and eventListeners.

We illustrate this analysis on the add method of Container class; Figure 4(left) shows excerpts of its Java implementation. We are interested in the side-effects this method has on the fields, discussed above, that are related to the abstract state defined in the previous section. Specifically, we are interested in side-effects on component field, which is used to implement containment properties. Figure 4(right-top) shows the output of BEG that encodes the results of side-effects analysis. BEG calculates that the method must side-effect the field component by assigning the parameter object to an element of the array. Unfortunately, the Java code is complicated by various checks on the method input and the state of the component field. If the array is too small, a new array is allocated, and the elements from the old array are copied to the new one. It is hard to design static analyses to accurately track such behavior, therefore, the analysis results may be overly imprecise. For example, the analysis, when unable to track exact values of integer-type variables, uses TOP\_INT value to represent any integer value. However, the model-writer can inspect the analysis results and decide whether to model the behavior that causes the imprecision. Suppose, we do not wish to model the allocation of the new array since any such error would be detected by simply executing the application, then the final model, shown in Figure 4(right-bottom), will reflect only the must side-effects of the method.

In addition to identifying methods that implement containment properties and registration of listeners (via various add methods), the analysis is able to identify methods that do not have any side-effects on the specified fields. A majority of methods in Swing only effect the look and feel of a GUI and thus have no side-effects on interaction order-related data. Once methods with no side-effects are identified, the model-writer can simply use the analysis results to model the behavior of these methods using empty stubs.

## 4 Customized Model Checking

In this section, we describe the modeling of GUI applications in Bogor’s input language, BIR, and the adaptation of Bogor’s state-storage strategy to reduce the time and memory requirements of analyzing GUIs with a single event-dispatching thread.

### 4.1 Modeling GUI Applications in BIR

BIR supports features found in the Java programming language. In fact, Bogor is being used in the next generation of the Bandera [5] toolset, which supports model checking Java programs by providing automated support for the extraction of safe, compact, finite-state models from Java source code. While BIR can naturally model Java programs such as Swing applications, it does not have primitives for non-deterministic choice expressions used in the models described in Section 3 (e.g., in Figure 3). Therefore, we define BIR extensions to implement these primitives.

Figure 5 presents the BIR extensions required by BEG. It declares an extension Choose which has four new non-deterministic expressions: (a) chooseBoolean() encodes a non-deterministic choice between true and false, (b) chooseInt( $i, j$ ) encodes a non-deterministic choice over integers in the given range  $i$  and  $j$  (inclusive,  $i < j$ ), (c) chooseObject $\langle\tau\rangle$ ( $subTypes, p_1, \dots, p_n$ ) encodes a non-deterministic choice over heap objects of type  $\tau$  (i.e., the type variable ‘rec\$ $\alpha$ ’ is replaced by  $\tau$ ) that satisfy predicates  $p_1, \dots, p_n$ , where  $subTypes$  indicates whether it should include sub-types of  $\tau$ , and (d) chooseReachableObject $\langle\tau\rangle$ ( $subTypes, o, f, p_1, \dots, p_n$ ) encodes a non-deterministic choice over heap objects of type  $\tau$  that satisfy  $p_1, \dots, p_n$  and are reachable from object  $o$  without reaching objects that satisfy the predicate  $f$ .

BIR supports side-effect free first-order functions similar to functional programming languages such as SML that can

```

extension Choose for bogor.ext.ChooseModule {
  expdef boolean chooseBoolean();
  expdef int chooseInt(int, int);

  expdef 'rec$a chooseObject<'a>(boolean, 'rec$a -> boolean ...);
  expdef 'rec$a chooseReachableObject<'rec$a>(boolean, Object, Object -> boolean, 'rec$a -> boolean ...);
}

fun isVisible(Component c) returns boolean = c.visible;
fun isEnabled(Component c) returns boolean = c.enabled;
fun isInvisibleComponent(Object o) returns boolean = o instanceof Component && !isVisible((Component) o);

(1) Choose.chooseInt(0, 3)
(2) Choose.chooseObject<JComponent>(true, isVisible, isEnabled)
(3) Choose.chooseReachableObject<JButton>(true, o, isInvisibleComponent, isVisible, isEnabled)

```

Figure 5. BIR Extensions: Non-deterministic Choice Expressions

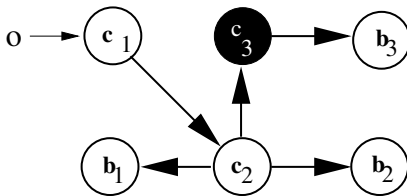


Figure 6. An Example Heap

```

package bogor.ext.ChooseModule
...
public class ChooseModule implements IModule {
  ...
  public IValue chooseBoolean(IExtArguments args) {
    IValue[] values = new IValue[] {
      vf.newIntValue(0), vf.newIntValue(1) };
    int index = ss.advise(..., values,
      args.getSchedulingStrategyInfo());
    return values[index];
  }
}

```

Figure 7. BIR Extension (excerpts)

be supplied for predicates  $p_1, \dots, p_n$  and  $f$ . For example, the function `isVisible` returns true if the given Component object is visible.

To illustrate, the BIR expression (1) in Figure 5 encodes a non-deterministic choice over  $\{0, 1, 2, 3\}$ . The BIR expression (2) encodes a non-deterministic choice over instances of `JComponent` (including sub-types) that are both visible and enabled in the state where the expression is evaluated (or returns the null value if there is no such instance). The BIR expression (3) encodes a non-deterministic choice over instances of `JButton` that are visible, enabled, and reachable through the object `o` without going through invisible Component objects. For example, suppose that we have the Java heap objects illustrated in Figure 6, where  $c_1$ ,  $c_2$ , and  $c_3$  are Components, and  $b_1$ ,  $b_2$ , and  $b_3$  are JButtons. Suppose that  $c_3$  is not visible, then evaluating (3) at that particular state will result in a non-deterministic choice between objects  $b_1$  and  $b_2$ .

Figure 5 illustrates syntactic extensions of the BIR language. Analogous to adding a new instruction in an interpreter, we also need to supply the semantics of the new expressions. In Figure 5, we declare that the Choose extension is implemented by the `bogor.ext.ChooseModule` Java class. Figure 7 illustrates how `chooseBoolean` is implemented using Bogor API.

In general, a BIR expression extension is implemented by a Java method with the same name (e.g., `Choose.chooseBoolean` is implemented by `ChooseModule.chooseBoolean`). When implementing non-deterministic choices over values as in Figure 5, there are two steps involved: (1) creating an array of values to choose from, and (2) submitting the array to the scheduler, which picks one value to return and records information to ensure that all other choices are subsequently chosen.

For example, the method `chooseBoolean` in Figure 7 creates an array consisting of false and true values which are represented by BIR integer values 0 and 1, respectively. It creates the values by using Bogor's `IValueFactory` module. Once the array of choices is created, it uses the scheduler (i.e., the `ISchedulingStrategist` module) to pick one of the values by invoking its `advise` method. The scheduler ensures that all values are considered by recording information about the choices that have been made so far in the scheduling information for the current state (i.e., the object returned by `args.getSchedulingStrategyInfo()`). This allows the scheduler to know whether there are remaining values to choose from, and if so which ones, when all successors of the current state have been explored. When implementing `chooseObject`, only the first step differs: traversing the heap to collect objects of the appropriate types that satisfy the given predicates to produce the value array. Bogor provides a visitor pattern API to traverse state and heap structures, thus, it is easy to implement extensions such as `chooseObject` and `chooseReachableObject`. The total code we wrote for GUI related extensions was less than 300 LOC using a typical Java code format.

## 4.2 Store-States-on-Choose Strategy

Bogor’s architecture is designed to ease customization of its module to exploit properties of an application domain to reduce the cost of model checking. We have customized Bogor to efficiently check the models of GUI applications generated by the techniques discussed in previous sections. More specifically, these models are single-threaded, reflecting the fact that Swing GUIs execute event-handling code in a single event-dispatching thread. Such models, even though a simplification of real implementations, allow for checking properties related to interaction orderings.

In contrast to general multi-threaded applications where interleaving may occur at each state, in GUI models with a single event-dispatching thread, branching in the state-space occurs only when choose constructs are used to model user selections or abstraction of the underlying application. Thus, we can reduce the number of states stored to those at which branching may occur and still preserve all user interaction orderings in the model.

Our solution is to modify the state storage strategy in Bogor to only store states in which a choose expression is invoked. The intuition is that those are the earliest points where we can decide whether the choices cause different states. There may be cycles in the state-space that are free of choose expressions, and to avoid non-termination of the search we must force the storage of states; we adopt the simple approach of setting a limit on the maximum number of transitions between each stored state. We implemented the algorithm described above in Bogor in about 30 LOC. We present data in the next section that illustrates the reduction achieved for reasoning about GUI applications.

## 4.3 Assessing Other Search Strategies

The strategy described above was just one of several that we explored for reducing the cost of model checking interaction order models of GUI applications. We implemented state-less search, where no states are stored, however, this proved inefficient as the number of GUI widgets grew; each widget requires a non-deterministic choice in the model and, consequently, the number of paths grows exponentially with the number of widgets.

We also considered a technique for reducing the memory requirements of model checking systems whose state-space has a cyclic structure [10]. A system is *quasi-cyclic* if after projecting the state-space onto a given set of state variables, one can find cycles. Repeated partial-states are defined by a predicate  $\Phi$ ;  $\Phi$ -conforming states define the boundaries of regions of the state-space. This allows us to divide the state-space search into a collection of individual searches of the regions. This technique showed excellent performance for design models of time-triggered embedded software [10].

Example	Measure	ALL	SSC
<i>Button Demo</i>	Trans	1920	2045
Objects: 50	States	1816	7
Choices: 3	Space (Mb)	40.2	39.6
Locations: 7563	Time (s)	4	0.8
<i>Voting Dialogs</i>	Trans	3114	4630
Objects: 120	States	2930	17
Choices: 4	Space (Mb)	45.5	44.5
Locations: 8269	Time (s)	10	1
<i>Dialog Demo</i>	Trans	88493	181512
Objects: 257	States	84439	1033
Choices: 14	Space (Mb)	74.3	47.6
Locations: 8689	Time (s)	512	38
<i>Calculator</i>	Trans	29016	35574
Objects: 362	States	27903	105
Choices: 24	Space (Mb)	66.4	48.6
Locations: 8789	Time (s)	183	20

Table 1. Experiment Data

GUI applications exhibit a quasi-cyclic structure: the event-dispatching thread in Swing has a loop that selects events, looks up their registered handlers, and then invokes those handlers. We characterized the event-loop header with a predicate and were able to apply our quasi-cyclic search. While this did yield a reduction in memory consumption over the unoptimized search, in all cases the reduction was less than that of the choose-only state storage strategy described above.

From this experience, we conclude that despite the fact that sophisticated reduction frameworks may exist in a model checking toolset, there may still be room for improvement. With a flexible model checking framework, in which alternative strategies can be prototyped and evaluated, one can exploit insights into the structure and behavior of a class of systems, such as GUI applications, to develop relatively simple yet effective reductions.

## 5 Model Checking Experience

We have implemented the techniques described in the preceding sections and applied them to a small collection of Java Swing GUI applications. These applications, while not as large as many real applications, contain a representative collection of Swing components. The source code and generated models for the sample applications are available online at <http://beg.projects.cis.ksu.edu>.

Table 1 presents the results of calculating the full state space of the applications on an Opteron 1.8 GHz (32-bit mode) with maximum heap of 1 Gb using the Java 2 Platform. For each example, we give the total number of *objects* allocated during system execution (nearly all of which are Swing component and container sub-types), the number of non-deterministic *choices* used to model user inputs, and the number of control locations in the combined model of

the Swing library, GUI implementation and underlying application. Due to abstraction of the underlying application the actual number of lines of code for an example can be many times larger than the number of locations.

In all runs, we used all of the reductions and memory-compression techniques available in Bogor (**ALL**) and compared that to the addition of the store-states-on-choose (**SSC**) strategy described in Section 4.2.

We note that the time to generate the models for these systems was negligible, except for the manual process of reading side-effects summaries for Swing methods and pruning them based on our understanding of their actual behavior. We did not keep detailed track of the time required for this part of the process, but it took several days to fine tune our model of Swing based on the large approximate starting model. Fortunately, that process happens only once and its cost can be amortized across the analysis of many Java Swing applications.

Our experiments do not demonstrate the benefit of our approach over simply submitting the Java application to a tool like JPF, but we note that none of these applications could be analyzed without some form of abstraction. The data state of the actual application, Swing component and container objects and the full-details of application and Swing methods results in enormous state-space explosion.

The data clearly show the benefit of customizing the analysis for single dispatch-threaded GUIs; reductions of more than an order of magnitude in run-time are achieved for the examples. We note that memory reduction is not as apparent since these examples are relatively small. We expect that as GUI implementations scale, especially in terms of the number of non-modal dialogs, significant memory reductions will be observed.

Finally, this data should be considered the *first step* in model checking GUI implementations. We have identified several additional opportunities for reducing the cost of model checking interaction orderings. For example, we are exploring the use of symmetries to collapse interaction orderings that pop up independent non-modal dialogs in different orders.

## 6 Related Work

An approach with goals similar to ours is implemented in GUI Ripper [16], which extracts a GUI model while executing a GUI system, using a DFS-like procedure to open windows, traverse events, and record information about the GUI state; the extracted model can be used for automatic test case generation. While the GUI Ripper builds models during execution of GUIs, our approach is based on static analysis which preserves information only about the logical state of the GUI.

There has been a long line of work on applying formal methods to reason about properties of HCI applica-

tions. Much of this work has focused on analyzing human-computer interaction orderings that can lead to a mismatch between the user's understanding of the state of the system and the state that is displayed by the computer system. The best studied of these ordering problems is *mode confusion* in aircraft autopilot systems [14, 19]. The analysis in this case operates on a hand built design or requirements model for the system. The intent is that once that model has been refined through repeated analysis and modification to be free of errors, it can be used as the basis for synthesizing GUI control logic.

Several researchers [2, 3, 17] have followed up on this idea to develop general methods for specifying the interaction behavior of user-interface systems, supporting the analysis of interaction specifications with model checking and other behavioral analyses, and synthesizing parts of GUI implementations. The benefits of starting with a high-level specification of interaction behavior is clear : it obviates the need for sophisticated model extraction techniques. Unfortunately, the synthesis techniques developed to date do not allow for the generation of fully-functional GUI implementations of the kind that are required in deployed HCI systems. Our work is an attempt to bridge this gap, by developing model extraction techniques that identify and preserve the semantics of the portions of the code that encode interaction behavior, while abstracting the rest of an application to enable tractable checking.

There has been relatively little work attempting to formally model and reason about GUI implementations. A notable exception is Chen's formal model of event-handling in Java GUIs [4], which effectively sets the stage for work such as ours. A related attempt is modeling web interactions [12], where a web server is modeled as a singly-threaded process capable of reacting to events sent by a singly-threaded client; despite model simplifications, the model allows for checking many characteristic and interesting properties of web applications.

Our previous work considered GUI implementations in the VisualBasic<sup>TM</sup> framework [8]. There we manually developed models for the framework, event-handler code and the application and used the SMV [15] model checker to reason about interaction ordering properties expressed as bounded safety properties in computation-tree logic. The work presented in this paper is the scaling and automation of our manual model extraction approach to treat Java Swing<sup>TM</sup> GUI applications.

Furthermore, we exploit the flexibility of the Bogor model checking framework to implement optimizations to the state-space search algorithm that exploit the structure of GUI applications. Our approach is similar to the work of Behrmann et al. [1], which explores the use of several heuristics to avoid state storage. They note that their heuristics do not work well in combination with depth-first search (DFS) algorithms. Our state-storage heuristic does work

well with Bogor's basic DFS because we can syntactically identify the points at which states should be stored.

## 7 Conclusions

A critical element of human-computer interaction is the control-logic that guides and restricts the set of user actions that can be executed at each system state. In modern Java Swing GUI implementations, this control-logic is spread across Swing framework class instances and user-defined event-handler methods. We have developed a static analysis framework that *extracts* that control-logic from Java source code and thereby enables exhaustive analysis of the behavior of a GUI implementation with respect to interaction ordering properties. Furthermore, understanding structural and behavioral properties of Swing GUIs has led to insight that has enabled domain-specific optimization of the analyses and the promise of further optimization. We have experimented with this analysis and our initial results are encouraging in both scaling to treat the features found in real Swing GUIs and in mitigating the potentially explosive growth of analysis cost. We are engaged in further study and experimentation aimed at understanding the range of correctness properties and the size of GUI applications for which this analysis approach will be cost-effective.

## 8 Acknowledgments

The authors would like thank the members of the Santos group at Kansas State University for many useful discussions about issues related to this paper.

The research reported in this paper was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607) by Lockheed Martin, by Rockwell-Collins, and by an IBM Corporation Eclipse Award.

## References

- [1] G. Behrmann, K. G. Larsen, and R. Pelanek. To store or not to store. In *Computer Aided Verification*, pages 433–445, Sept. 2003.
- [2] J. Berstel, S. C. Reghizzi, G. Roussel, and P. San Pietro. A scalable formal method for design and automatic checking of user interfaces. In *Proceedings of the 23rd international conference on Software engineering*, pages 453–462, 2001.
- [3] J. Campos and M. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 3(8):275–310, Aug. 2001.
- [4] J. Chen. Formal modelling of Java GUI event handling. In *Formal Methods and Software Engineering : 4th International Conference on Formal Engineering Methods*, Oct. 2002.
- [5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
- [6] X. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, and Robby. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*, Nov. 2002.
- [7] M. B. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
- [8] M. B. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 244–261. Springer-Verlag, Sept. 1997.
- [9] M. B. Dwyer, J. Hatcliff, V. Ranganath, and Robby. Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. *Formal Methods in System Design*, 2004. (to appear).
- [10] M. B. Dwyer, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proceedings of the 3rd International Conference on Embedded Software*, Oct. 2003.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub. Co., Jan. 1995.
- [12] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In *In Proc. 12th European Symposium on Programming, Lecture Notes in Computer Science, Warsaw, Poland, Apr. 2003*. SpringerVerlag., 2003.
- [13] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
- [14] G. Lüttgen and V. Carreño. Analyzing mode confusion via model checking. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking (SPIN '99)*, volume 1680 of *Lecture Notes in Computer Science*, pages 120–135, Toulouse, France, September 1999. Springer-Verlag.
- [15] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [16] A. M. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering (WCRE '03)*, pages 260–269, 2003.
- [17] F. Paterno and C. Santoro. Integrating model checking and HCI tools to help designers verify user interface properties. In *Interactive Systems - Design, Specification, and Verification : 7th International Workshop*, June 2003.
- [18] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
- [19] J. Rushby. Using model checking to help discover mode confusions and other automation surprises. In D. Javaux, editor, *Proceedings of the 3rd Workshop on Human Error, Safety, and System Development (HESSD '99)*, June 1999.
- [20] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 188–197. ACM Press, 2003.
- [21] O. Tkachuk, M. B. Dwyer, and C. S. Păsăreanu. Automated environment generation for software model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, Oct. 2003.
- [22] W. Visser, G. Brat, K. Havelund, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, Sept. 2000.