AMC: Verifying User Interface Properties for Vehicular Applications

Kyungmin Lee*, Jason Flinn*, T.J. Giuli†, Brian Noble*, and Christopher Peplin†
University of Michigan* Ford Motor Company†

ABSTRACT

Vehicular environments require continuous awareness of the road ahead. It is critical that mobile applications used in such environments (e.g., GPS route planners and location-based search) do not distract drivers from the primary task of operating the vehicle. Fortunately, a large body of research on vehicular interfaces provides best practices that mobile application developers can follow. However, when we studied the most popular vehicular applications in the Android marketplace, no application followed these guidelines. In fact, vehicular applications were not substantially better at meeting best practice guidelines than non-vehicular applications.

To remedy this problem, we have developed a tool called AMC that uses model checking to automatically explore the graphical user interface (GUI) of Android applications and detect violations of vehicular design guidelines. AMC is designed to give developers early feedback on their application GUI and reduce the amount of time required by a human expert to assess an application's suitability for vehicular usage. We have evaluated AMC by comparing the violations that it reports with those reported by an industry expert for 12 applications. AMC generated a definitive assessment for 85% of the guidelines checked; for these cases, it had no false positives and a false negative rate of under 2%. For the remaining 15% of cases, AMC reduced the number of application screens that required manual verification by 95%.

General Terms

Human Factors,Verification

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*

Keywords

Model checking; Android; Vehicular user interfaces

MobiSys'13, June 25-28, 2013, Taipei, Taiwan

1. INTRODUCTION

Some mobile applications are often used in environments in which the user's attention must be focused on driving. For instance, drivers employ GPS route planners, music services, gas station locators, parking-spot finders, and similar tools. While well-designed applications that minimize driver distraction can improve the driving experience, poorly designed applications can too easily divert attention from the primary task of operating the vehicle. Although vehicular applications are currently the primary reason why mobile interfaces are currently designed with user distraction in mind, emerging technologies such as augmented reality are likely to increase the prevalence of such situations.

Unfortunately, as we show in a study of Android applications, many tools intended for vehicular usage today do not follow accepted industry guidelines. We argue that this problem does not arise from a lack of usability research; in fact, there exists a large body of detailed guidelines and standards for user interaction in vehicular environments [7, 10, 16, 19, 21]. Rather, the issue is that most mobile application developers simply do not have enough experience creating interfaces for automotive environments. They are not aware of the best practices in vehicular interface design, and they are not able to anticipate how drivers will interact with their applications in vehicular settings. This information gap results in applications that inappropriately demand too much attention from drivers.

Currently, there are several models for running mobile applications in vehicles. Some applications run on an in-vehicle *human-machine interface* (HMI) such as MyFord Touch or the Cadillac Cue system. Access to such platforms is closely controlled by vehicle manufacturers; any applications developed by third parties undergo strict testing in cooperation with manufacturers before they are deployed. Usability experts ensure that applications meet guidelines for in-vehicle use on a manufacturer's HMI platform. The advantage of this approach is that deployed applications meet a very high usability standard. The disadvantage is cost and speed of deployment; application testing can take many iterations between developers and usability experts. Developers must wait for feedback after each modification to the application interface, and each application evaluation requires a substantial amount of costly expert time.

A more recent class of in-vehicle applications run on smartphones. These applications are deployed through application marketplaces without the involvement of vehicular manufacturers. The advantage of the smartphone model is speed of deployment; mobile developers can often release a new application months after conceiving an initial idea. The disadvantages of this approach are two-fold. First, the input and output capabilities of a smartphone platform are not ideal for the vehicle; in contrast, vehicle HMIs

have dashboard touchscreens, steering wheel controls, and other forms of interaction specifically designed for use in a vehicle. Second, as previously mentioned, mobile application developers are unaware of best practices for vehicular applications and often make poor design choices that lead to too much distraction when the application is used.

Recently, a hybrid model of application deployment has emerged in which an application running on a smartphone seamlessly interacts with in-vehicle HMIs by, for example, displaying the application graphical user interface (GUI) on the dashboard touchscreen and by enabling input via steering wheel controls. Ford's AppLink platform and the MirrorLink standard are two examples of this hybrid approach. While such platforms are a promising method for addressing the input and output limitations of smartphones, they provide little help to developers struggling to design applications appropriate for vehicular environments. Even if expert guidance were available to such developers, it is infeasible to have completely-manual testing scale to check the total number of vehicular applications in a marketplace such as iTunes or the Android store.

In this paper, we mitigate this dilemma with automated model checking of user interfaces. We have developed a tool, called AMC (Android Model Checker) that automatically explores the set of GUI screens displayed by an Android application and verifies that desired user interface properties hold over all such screens. AMC does not require application source code, nor does it require any application-specific knowledge. Instead, AMC develops a model of the application's GUI by executing the application. AMC regards each unique screen as a state, and it considers UI events (e.g., button presses) as transitions between those states. It explores the application state space (the GUI), until it has exhausted all reachable state transitions. It verifies properties such as minimal use of animation, appropriate color contrast, button closeness on the touchscreen, and task complexity as measured by the number of user actions required to complete an action. AMC outputs a list of violations of best practice guidelines for these properties along with snapshots that detail the specific violations.

The goal of AMC is not to completely eliminate the need for testing by human experts. Some issues defy automated judgment and are difficult to quantify. For instance, most experts would consider a moving graph of instantaneous gas mileage to be acceptable for vehicular usage, but would clearly consider a game with animation to be unacceptable. Automatic differentiation of the two animations can be difficult.

Instead, the goal of AMC is to substantially reduce the amount of work that a human expert must perform to help developers verify an application. AMC can check "easy" properties automatically, eliminating such tasks entirely from the expert's purview. For more challenging properties, AMC can prune large numbers of states and identify a small set of questionable screens that an expert must check manually. This helps the expert focus her time on the tasks that most require human judgment.

AMC also helps application developers by providing them with early feedback about usability violations. A developer can run AMC, generate a list of best practice violations, and fix those bugs without submitting the application for expert testing. After the application passes all AMC automated checks, expert checking can reveal any remaining problems.

To evaluate our tool, we studied 13 popular applications in the Android marketplace: 8 of these applications are designed for vehicular usage and 5 are not. Unsurprisingly, we found that applications that are not specifically intended for vehicular usage often do not follow accepted industry guidelines for vehicular interface design. However, we were surprised to find that *the Android applications intended for vehicular usage violated best practice guidelines at approximately the same rate as non-vehicular applications*. These results highlight the critical need for tools such as AMC that can help developers understand why their user interfaces may be inappropriate for use by the driver of a vehicle.

We evaluated AMC by comparing its results to those manually generated by an industry expert checking the same applications. AMC generated a definitive assessment for 85% of the application/property combinations that we studied. Compared to the expert assessment of these combinations, AMC had no false positives and a false negative rate of less than 2%. AMC deferred the remaining 15% of application/property combinations for further assessment by a human expert, but it reduced the number of screens that an expert must check in such cases by 95%. The median time for AMC to completely explore an application GUI is approximately 73 minutes; the most complex application that we studied (Twitter) required approximately 10 hours.

2. OVERVIEW

AMC automatically explores the GUI of an Android application. It checks the interface for violations of a set of pre-specified design properties. AMC uses a model-checking technique that attempts to exhaustively explore all screens that can possibly be displayed by the application. For each property, it may report that the property is upheld by the application (no violations were found), or it may output a list of violating screens annotated with screen snapshots that demonstrate each violation. Occasionally, AMC may be unable to automatically determine whether a property is upheld on a particular screen; in such cases, it reports a possible violation with an annotated snapshot.

While the properties that AMC checks are pre-configured, AMC contains no application-specific logic. Testing is fully automatic once started, so a user can leave AMC running and come back later to collect results. For applications with a cloud component, AMC requires that the testing computer have a working Internet connection; AMC also requires that any needed authentication be completed before testing starts. These requirements are minimal so as to make it trivial to apply AMC to testing a new application.

We envision two classes of users. Applications developers will use AMC to get early feedback on the distraction properties of their application GUI. A developer can run AMC in his own testing environment and get valuable insights without having to wait for feedback from a human expert.

We also envision that experts who control access to a vehicular application marketplace will use AMC to magnify their productivity. Many properties such as a maximum number of user actions per task and a minimum button size are tedious to check manually on each application screen, yet these properties are easy for a computer to verify automatically. We expect that an expert will simply rely on AMC to check these types of properties. Other properties, such as the acceptability of animation, require human judgment. AMC can classify some simple cases; for the remaining cases, it enumerates the specific screens that the expert should check further. Thus, even for properties that are hard to verify automatically, AMC can still substantially reduce the testing time required of a human expert.

3. DESIGN AND IMPLEMENTATION

AMC verifies user interface properties with a model checking approach similar to that used by CMC [15] in which it directly checks the implementation of an application. AMC starts with no information about the user interfaces states and the transitions among those states; instead, it discovers this information on the fly by triggering user actions and observing their effect on the application GUI.

Most prior model checkers operate by taking checkpoints of application states to allow the checker to deterministically revert to any past state by restoring a checkpoint. We initially planned to use this approach in AMC, but we quickly realized that checking mobile applications presents a substantial complication: the client typically only contains a portion of the application state, with the remainder contained in a cloud component. Restoring only the mobile component to a prior state is infeasible because the mobile and cloud components become out-of-sync. It is rare to see a mobile application in which the interaction between client and server is truly stateless (for instance, encrypted connections have inherent state, caching introduces state dependencies, and even simple identifiers often have constraints such as increasing monotonically).

AMC could potentially take a coordinated checkpoint on the mobile client and the cloud server. This is infeasible for third-party testers who do not have access to the cloud component. Even developers may lack the ability to checkpoint cloud state if the cloud component runs in an environment hosted by another organization.

For these reasons, AMC instead explores application state without the use of checkpoints. It uses the application GUI to return to desired states. This requires that AMC use its model of states and transitions among those states for navigation. It also requires that AMC discover when a new transition leads to a state that it has seen previously. This section discuss how AMC handles these and other challenges.

3.1 Modeling the user interface

AMC represents the graphical user interface of an Android application as a finite-state model. Intuitively, each distinct screen of the application is a state, and the user actions that transition between those screens are state transitions. AMC's model is therefore a directed graph in which the distinct screens are vertexes and the user actions are edges. In practice, the notion of what represents a "distinct" screen is open to interpretation, and AMC uses state equivalence heuristics, described in the next section, to determine whether two screens with different content and structure should be classified as the same state or as different states.

AMC constructs its state model on the fly. It begins with a single state that represents the initial screen displayed by the application. AMC adds a transition for each user action that may be performed on that screen such as a GUI button. Each of these transitions is initially marked as being unexplored.

AMC's basic state exploration algorithm works as follows. If there are no unexplored transitions left in the state model, AMC has fully explored the application GUI, and so it terminates. Otherwise, AMC chooses an unexplored transition. It orders the unexplored transitions by the number of actions required to reach that transition from the current application state. For instance, unexplored transitions on the screen currently being displayed are given top preference, transitions on any screen that can be reached using one user action are given the next highest preference, and so on. AMC determines the number of user actions to reach a particular state by executing a breadth-first search on the graphical representation of the state model starting at the current application state and terminating when the first unexplored transition is encountered.

After choosing an unexplored transition, AMC generates user events to navigate to the screen containing that transition (unless the transition is located on the current screen, in which case no navigation is required). AMC uses the Android monkey package [24] to send user actions such as touchscreen events and button presses to the Android screen. AMC determines the screen coordinates for such events by parsing the DOM tree, which describes all currently displayed widgets and their relative locations.

AMC next performs the user event necessary to effect the transition and observes the effect on the application GUI. Some actions do not cause the application to display a new screen; AMC models such actions as explored transitions that start and terminate in the same state. Some actions transition to a state that has already been observed by AMC (we defer the discussion of how AMC determines state equivalence to the next subsection). AMC simply adds the explored transition as a vertex from the old state to the new one. The remaining transitions generate a state previously unseen by AMC.

When AMC discovers a new state, it adds the state to its model, marks the transition it just took as explored, and sets the destination of the transition to the new state. It then takes a snapshot of the current screen using the screen capture feature of the Dalvik Debug Monitor Server, which is part of the Android SDK library. Snapshots are saved and may later be used to explain observed violations to the user. Next, AMC verifies user interface properties, as described in Section 3.5. Some properties, such as button size, pertain to individual states. For these properties, AMC highlights the violating region on the screen snapshot for that state in its report. Other properties, such as the number of user actions required to perform a task, are a property of multiple states and transitions. AMC generates a series of snapshots that illustrates the violation in such cases. We have found annotated snapshots to be an effective mechanism for quickly communicating violations to the user of our tool.

3.2 Determining state equivalence

AMC's model checking algorithm requires a good metric for determining state equivalence. To understand why this is crucial, consider a straw man equivalence function in which two GUI screens are considered to be the same state if and only if they have the exact same content.

AMC uses the Hierarchy Viewer tool [12] to extract the DOM information, expressed in XML format, for the currently displayed Android screen. The DOM tree contains all of the GUI widgets on a given screen along with any content within those widgets. For instance, the DOM tree will have a node for a list box element, as well as a node for each list item that describes the item contents. AMC generates a unique identifier for each screen, called the *content hash* by traversing the DOM tree in depth-first order and computing a hash over all nodes. Thus, AMC could potentially define two screens to be the same state if they have matching content hashes and declare them to be different states if their content hashes do not agree.

However, the proposed state equivalence function works poorly for most applications of reasonable complexity. Even simple actions, such as adding an item to a list or changing displayed text for an object will lead to a new state. For applications with cloud components, simply refreshing the same screen and downloading new content will lead to a new state. As a result, AMC state exploration will never terminate. Even worse, AMC may spend unnecessary time exploring "new" states that are essentially identical to states that it has seen previously. Thus, strict content equivalence is a poor choice for a state equivalence function.

Ideally, we would like to define state equivalence to match a user's intuitive notions of what constitutes the unique screens in an application. For instance, we would prefer that AMC treat the business and sports section screens of a newsreader application as the same state since both simply provide a list of articles that can be accessed. AMC therefore uses a state equivalence function that is based on structure, not content.

When a screen is displayed, AMC computes its *structure hash* by traversing the DOM tree in depth-first order and hashing the tag name, level, and sibling order of each XML node, omitting all other content. AMC uses the structure hash as a unique identifier for each state. Thus, if two screens have the same structure (layout of buttons, lists, text fields, custom views, etc.), but the screens have different content, AMC considers the two screens to be the same state.

From our initial experience using AMC, we learned that the structure hash is still too strict an equivalence check. As defined so far, the structure hash considers two screens to be different if a list or other container has a different number of elements. For instance, a screen that displays 3 gas stations would not be equivalent to another screen that is identical except for displaying 4 gas stations. This problem arises because individual list items are XML nodes and therefore affect the structure hash. Unfortunately, if this problem is not addressed, the AMC algorithm will sometimes fail to terminate (this is the familiar "state explosion" problem in model checking [25]).

On the other hand, we do not want to omit considering such list items entirely. Having one or two items in a list often enables actions that are not accessible if no items are in the list (for instance, one cannot click on a list item if none are present). Further, some lists such as menus do not contain items of homogeneous type; clicking on different items will have different effects.

AMC uses the following heuristic to handle lists and similar containers. If a list has more than one item, AMC checks whether those items are homogeneous. If the list items have registered different callback handlers, then they are considered heterogeneous (since clicking on different items is likely to have different effects). Since AMC does not require application source code, it distinguishes between two callback handlers by using the binary callback address passed to the GUI system.

Some applications multiplex many actions through the same handler. Therefore, if the callback handler is the same, AMC clicks on the first 2 items. If the items transition to different states, then the list items are deemed heterogeneous; otherwise, they are homogeneous. When AMC sees a list with more than 2 homogeneous items, it canonicalizes that state by considering only the first 2 items in the structure hash. This has the effect of making all instances of the same screen with two or more list items be the same state. Screens with different numbers of heterogeneous elements are always considered to be distinct states.

3.3 Non-deterministic transitions

From the point of view of AMC, user actions can sometimes appear to be non-deterministic. That is, taking the same action from the same state transitions to different states at different times. There are several reasons why this behavior can arise. In rare cases, the application may be truly non-deterministic. More often, the state transition is deterministic, but the resulting state depends on some aspect of application state that AMC cannot observe because it is unaware of application semantics. For instance, the back button may transition to a different state depending on the path that the user took to get to the current state. In another example, changing a configuration option in a menu may cause an action to transition to a new state. Mistakes in state classification can also lead to seemingly non-deterministic behavior. For instance, if AMC classifies two different screens as belonging to the same state, it is possible that performing the same action on each screen leads to a different state. While such misclassifications are rare, they will inevitably occur in any model-checking system like AMC that uses heuristics to avoid state explosion.

Seemingly non-deterministic behavior complicates the basic search strategy described in Section 3.1. When AMC performs user actions to transition to the nearest state with an unexplored transition, it may go astray and arrive at a different state than expected. Therefore, AMC must enhance its basic search algorithm to deal with such behavior.

AMC considers all transitions to be deterministic unless proven otherwise. AMC maintains a second graph of the application state model, called the *exploration graph* — this graph contains only deterministic edges. New edges are added to the exploration graph when AMC first explores the transition. However, if AMC retraverses an edge in the exploration graph and the transition leads to a state other than the one expected, it labels the transition as non-deterministic and removes the edge from the exploration graph.

AMC calculates the path to the nearest unexplored transition by applying the breadth-first search algorithm described in Section 3.1 to the exploration graph. Effectively, this finds the shortest path that traverses only deterministic edges. When the path traversal fails due to a newly-discovered non-deterministic edge, AMC re-runs the shortest path algorithm after the edge has been removed.

If too many edges are removed due to non-determinism, AMC may not be able to explore some transitions because it can no longer find a deterministic path to the state that contains the transition. However, we have observed that most seemingly non-deterministic transitions are fully deterministic if the application performs an identical sequence of n user actions (where n is greater than one). As a simple example, the back button often transitions to the state explored prior to the current state. Thus, clicking the back button on a given state appears to be non-deterministic if there exists two or more possible prior states. However, if one considers a 2-step sequence of actions, i.e., transitioning from the prior state to the current state and then clicking the back button, then the transition as a result of that sequence appears completely deterministic.

If after exploring all transitions that can be reached via deterministic transitions, AMC still has an unexplored transition, it uses the following algorithm to try to reach the state containing that transition, which we refer to the goal state. AMC keeps a log that contains the sequence of all states that it has visited along with each transition that it has triggered. It searches through the log to find all occurrences of the goal state. For each occurrence, it observes the past n actions that it took to reach the goal state (n is initially 2). It assumes this sequence is deterministic unless proven otherwise. So, it inserts in the exploration graph an edge from the starting state of the sequence to the goal state. The edge is given a weight of n since it contains n user actions. The edge remains in the graph until the sequence is demonstrated to be non-deterministic. AMC then restarts exploration and tries to reach the goal state via the new paths that it has added. If AMC again fails to reach the goal state, it tries again with $n = 4$ and $n = 8$. After $n = 8$, AMC gives up on exploring this particular transition and reports the state containing the transition as being only "partially explored". If there are multiple unexplored transitions, AMC proceeds until it either has explored or has given up on all such transitions.

3.4 Complications in state exploration

The Android platform has a few peculiarities that complicate AMC state exploration. First, Android does not provide a callback to an external observer when the display of a screen completes. Therefore, AMC has no means of knowing when to compute the structure hash. Initially, we used a simple timeout approach in which AMC waited 10 seconds for each screen to stabilize before computing the hash. However, this strategy substantially increased AMC's state exploration time since AMC waited after every transition. We could not reliably set the timeout to a lower value because some slow applications take several seconds to fetch data from a remote server and display results.

AMC currently uses a modified timeout-based approach. After exploring a transition, AMC continuously computes the new screen's structure hash. If the hash matches that of a previously explored state, then AMC considers the screen to have stabilized and waits no longer. With this approach, AMC waits for the full 10 second timeout only once for each new application state, and not once for every transition. This substantially decreases state exploration time because the number of states is much less than the number of transitions.

Many Android application screens have components that allow users to scroll down to view more items than can be displayed at one time. Depending on the specific method used by developers to implement scrolling, items not currently displayed may or may not appear in the DOM tree. For example, elements not currently displayed appear in the DOM Tree when the Android ScrollView widget is used, but not when the ListView widget is used. In the former case, AMC does not need to perform any extra actions. In the latter case, AMC sends events to the screen to cause the application to scroll down until all elements have been displayed. AMC identifies new elements that appear during scrolling by their unique row identifier; it adds these elements to its internal representation of the screen's DOM tree. Once the DOM tree is complete, AMC proceeds as described previously.

Some applications require text input to operate. The two examples we encountered were passwords and destination fields for GPS applications. In such cases, the AMC user must supply the appropriate text to enter and the associated text field's id prior to running AMC. With this information, AMC automatically inputs the supplied text on user's behalf whenever it encounters the associated text field on a screen.

3.5 Verifying user interface properties

User interface guidelines for vehicular settings are based on two key principles. First, vehicular application should not give rise to potentially hazardous behavior by the driver or other road users [21]. The driver's primary focus must be on the road and the driving task; the vehicular application's user interface should minimize its distraction level. This principle bounds the amount of time that the driver can spend on each action. For instance, as button size decreases, Fitts' Law [9] predicts that it will take longer for the user to position her finger to touch the button. Thus, overly small buttons require too much attention and distract driver focus from the road.

Additionally, actions should not have time limits since a driver should not be forced to divert his attention from the road at inappropriate moments. Many forms of animation violate this principle, since a user must watch the screen at a particular time to gather needed information.

Another principle is that that application must be consistent and limit state that the driver has to remember [10]. Being consistent includes maintaining the general look, feel, and layout of the application. For example, buttons should be placed in the same portion of the screen, with the same general functionality, so that the user can easily navigate. Limiting state bounds the maximum number of actions that can be required to perform a task.

AMC currently verifies seven properties: number of actions to perform a task, text contrast ratio, word count per screen, button size, button closeness, use of animation, and the amount of scrolling required per screen. AMC presents a modular interface for adding new properties, and it separates the verification code from the state exploration code. This design has made it easy to add new property verification modules as our experience with the tool has grown. We next discuss each property that AMC verifies in more detail.

3.5.1 User actions per task

Most vehicular interface guidelines suggest limiting the number of user actions (button presses, touchscreen events, etc.) that a driver must perform in order to complete a single task [7, 16, 19]. Excessive sequences of user actions increase the cognitive load on the driver and divert too much visual attention from the road. Since existing guidelines do not agree on a specific maximum number of actions that should be allowed, we consulted with an expert in Ford Motor Corporation's HMI lab, who suggested that no task should take more than 10 actions to complete.

While the notion of a task is somewhat application-specific, AMC can generate reasonable approximations of tasks using the graphical model created by state exploration. Given a starting state, a task is a series of actions that transitions to some destination state and then returns back to the starting state. For instance, a task could be navigating through menus and sub-menus, setting a configuration option, and then returning via the back button.

For every state in the graph, AMC finds the shortest round-trip path to and returning from every other state. If the length of this path is greater than the limit of 10 actions, AMC reports a violation. It appends annotated snapshots of all screens along the violating path and the transitions that move from one screen to another. This check is performed after state exploration finishes. Detecting violations does not require revisiting states since all necessary information is contained in the graphical state model.

AMC exhaustively considers all tasks rather than attempting to identify the set of tasks that are performed frequently. Such identification would require application-specific knowledge that AMC does not possess. Further, even infrequent tasks that violate guidelines should be avoided.

3.5.2 Text contrast ratio

Drivers should be able to discern screen elements with a quick glance. A poor contrast ratio between elements displayed on a screen increases the amount of time and attention required to understand the screen contents. Consequently, a contrast ratio of at least 3:1 between the luminance of the foreground and background is recommended [19]. AMC currently verifies this property for all text elements displayed by an application.

AMC uses the snapshot of each state to calculate contrast ratios for text elements. It parses the DOM tree to find all view objects that have text elements. It then extracts color information and screen locations for all such objects. AMC computes the background and foreground RGB values for each element, and it converts these values into luminance values using a conversion formula proposed by W3C's Web Content Accessibility Guidelines [6]. If the contrast ratio is below 3:1, AMC reports a violation and highlights the violating element on the state snapshot.

3.5.3 Text word count

Excessive text on a screen requires too much driver attention. According to industry-standard user interface guidelines [7], a task should take no longer than 20 seconds to finish. The average adult reads approximately 250 to 300 words per minute [27]. Thus, no screen should ever contain more than 100 words. Using this as a conservative upper bound (it is trivial to set the threshold to lower values), AMC counts the number of words in all text elements and reports a violation if there are more than 100 total words for any state.

Applications frequently embed text in images (for example, by displaying logos or using buttons with bitmap images). When text is embedded in an image, the DOM tree does not provide a description of that text. AMC handles these cases with Optical Character Recognition (OCR). When AMC encounters a leaf in the DOM tree that does not have a text element, it first calculates the screen region for that element. It then uses the Tesseract OCR tool [22] to identify any text within that region. If AMC discovers text, it adds the word count for that text to the total word count for the state.

If the screen has a scrolling component that hides some elements from view, AMC scrolls down to display more elements. This continues until all elements have been displayed or until the word count exceeds the specified limit.

3.5.4 Button size

Initiating touchscreen events should not be excessively distracting. If the surface area that initiates an event is too small, the driver must focus her attention on making sure that her finger contacts the screen at exactly the right location; this diverts attention from the road. Consequently, best practice guidelines say that the minimum contact surface area for a control such as a button is $80\,\text{mm}^2$ [19].

AMC verifies this property by first determining the height and width of a pixel on the test platform. In our evaluation, we assume a 9-inch dashboard touchscreen. This is conservative for Android applications since most smartphone screens are substantially smaller. For each clickable item on the screen, the DOM tree reveals the height and width of the item in terms of the number of pixels. By multiplying these values and scaling by the platform-specific conversion factors, AMC calculates the absolute button size. If this is less than $80\,\text{mm}^2$, it reports a violation and highlights the button on the application screen snapshot.

Android allows applications to add their own window element types; these application-specific types are referred to as *custom views*. For instance, many vehicular applications use a custom view to display a map element. Custom views often contain clickable button-like elements. Unlike standard buttons which are shown as elements in the DOM tree, individual clickable elements within an opaque custom view are hidden from AMC. AMC can detect the existence of a click handler function, but it cannot determine the size or location of clickable regions from the DOM tree. It cannot even determine the number of clickable regions in the custom view.

AMC uses empirical testing to learn more about clickable elements in a custom view. It sends click events to a custom view at $< x,y >$ coordinates that are laid out in a grid pattern such that the horizontal and vertical difference between clicks is slightly less than 50% of the minimum button size given a square button. For our 9-inch touchscreen, this works out to a grid with coordinates 20 pixels apart.

AMC then clicks on each grid coordinate. If the click causes the application to transition to a different state, AMC records the state at which it arrived, then returns to the original state to click on more grid coordinates.

AMC considers any set of contiguous grid coordinates that transition to the same state to be part of the same clickable region (button). For simplicity, it currently assumes that such regions are rectangular. Because the grid is regularly spaced, AMC can bound the minimum and maximum dimensions for a region. For instance, a region that is two clicks in width must have width between 21 and 59 pixels. AMC calculates the minimum and maximum button area from this information. If the maximum area is less than $80\,\text{mm}^2$, a violation is reported. If the minimum area is at least $80\,\text{mm}^2$, no violation exists. Otherwise, AMC needs more information to make a determination, since the button area could either be bigger or smaller than the minimum area. It therefore clicks on additional $< x,y >$ coordinates in a binary search pattern to determine the height and/or width of the region. The search terminates once a violation is detected or the button is determined to be at least as large as the guidelines require. In other words, AMC narrows its estimate of the button size until it can precisely determine whether or not the button violates the guideline.

This method of determining region size relies on the custom view not changing between clicks. Once the view changes, button locations may be added, removed, or moved; in other words, AMC can infer no useful information about region size. AMC checks to see if a custom view changes by comparing screen snapshots of the view region. It takes an initial snapshot before sending any click events. Five seconds after each click, it takes a new snapshot of the view region and compares that with the original snapshot. If the snapshots differ by more than 1%, AMC declares the view to have changed. In this case, AMC reports the view as a potential violation that requires further analysis because it cannot make a definitive judgment about the clickable regions contained within the view.

3.5.5 Button distance

For reasons similar to those cited for button size, the distance between the center of each pair of buttons should be at least 15 mm [19]. AMC determines the geographical center of each button on the screen, calculates the Euclidean distance in terms of the number of pixels, converts that value to a screen-specific absolute distance value, and reports a violation if the limit is exceeded.

For custom views, AMC uses a similar strategy to the one used to determine button size. It first uses the grid information to determine bounds on the center of each clickable region (again, assuming that such regions are rectangular). If all regions can be determined to be more than the maximum distance from each other, AMC terminates without reporting a violation. If any pair of regions are definitely too close to each other, AMC reports a violation. If the grid has insufficient granularity to determine whether a pair of regions are too close, AMC performs binary search to refine the bounds on the center of each button until it can determine whether or not a violation exists.

3.5.6 Animation

Drivers must be free to interact with the application at a time of their convenience. No application should ever demand that an action be performed within a time limit, since performing the action may force the driver to divert attention form the road at an inappropriate instance. A corollary of this guideline is that an application should never display animations, videos, and other highly-interactive visual elements [7].

Unlike the previous guidelines, human judgment is usually required to determine whether a visual artifact is acceptable. For instance, an animated graphical display of fuel efficiency in a hybrid vehicle is typically considered to be acceptable. On the other hand, interrupting a route display in a GPS application to pop-up an unsolicited chat message from a nearby vehicle would almost cer-

tainly be considered poor vehicular interface design (note: this is an actual application behavior that we observed!). AMC therefore detects *potential* violations of this property, but defers the actual evaluation to an expert tester. The goal of checking this property is therefore to substantially reduce the number of states that such an expert needs to examine.

When AMC discovers a new state, it pauses for four seconds to detect possible animations. During this interval, it takes several screen snapshots. It breaks each of the screen regions into blocks using a grid pattern and calculates the Sum of Absolute Differences (SAD) on each color channel to determine if a block has changed significantly between snapshots. Any block that has SAD value more than zero on any color channel is considered a potential violation. The changing regions on the screen snapshots are highlighted to illustrate the potential violation.

If an animation only affects a small portion of the screen, it is unlikely to be distracting. For instance, a blinking cursor can be helpful in drawing attention to an important screen region and may help reduce cognitive load.

When AMC detects animation on a screen, it divides the screen into 400 equally-sized blocks. It then compares each of these regions from the two snapshots. If content has changed in less than 10% of the blocks, AMC determines that there is no violation. If more than 10% of the content has changed, AMC reports the state as a potential violation. This rule substantially reduces the number of states that need to be manually examined and has not led to false negatives in any application.

3.5.7 Scrolling

Scrolling through many options commands too much attention from a driver. Usability guidelines therefore recommend eliminating scrolling or limiting the number of items in a list [19]. Alternative selection approaches such as voice recognition can be used when options are too numerous to include in a scrolling control.

AMC verifies that no list element contains more than 10 elements — we chose a limit of 10 based on a recommendation from an industry expert. For each list view object in a state's DOM tree, AMC reads the value of the 'mItemCount' field. If this value exceeds the limit, AMC reports a violation.

4. EVALUATION

We evaluated AMC by comparing its results to those of an industry expert evaluating 12 popular Android applications. We also calculated metrics for AMC such as state coverage and exploration time. Finally, we assessed the need for a tool such as AMC by examining how often current Android applications intended for vehicular settings violate best practice interface design guidelines.

4.1 Setup

We executed AMC on a Nexus One phone running Android 2.3.4. We have modified our Android system to enable AMC to extract event handler and text color information.

We assumed the existence of a phone-HMI hybrid interface that replicates the smartphone's display on a 9-inch vehicle touch screen. We further assumed that all user interactions with the application will take place using the vehicular touch screen.

Note that a 9-inch display is larger than that of almost all Android phones. Therefore, buttons and icons will be significantly larger on the touch screen. Any size-related violation we identify assuming the existence of a touch screen would also be a violation on most any Android phone.

Some applications support multiple input modalities for initiating an action, for example, UI events and voice commands. We check the UI in such instances because some set of users will opt for touch screen interaction, and so such interaction should meet best practice guidelines.

We selected 14 applications for our evaluation. These were selected because they were the most popular applications in their respective categories in the Android store as of November 2011. Five applications (Gmail, Google Finance, MyDays, eBay, and Twitter) are not targeted for use in vehicular environments. We expected that such applications would have numerous usability violations because they have not been designed for vehicular usage.

We defined eight of the applications (GasBuddy, Google Maps, Google Navigation, TomTom Navigation, Waze, iRadar, Beat the Traffic, and Best Parking) to be *vehicular applications*. Informally, a vehicular application is one that delivers information that may be particularly relevant to the driver of a vehicle, such as gas prices at nearby stations, recommended routes, police radar locations, and traffic information. Some applications, such as Waze, have specific mechanisms such as voice control for minimizing driver distraction. Although other applications, such as Google Maps, are not solely intended for use by a driver, our experience is that many people use such applications while driving because of the helpful information they provide. We expected that these applications would have considerably less violations than the first set (although our expectation turned out to be wrong).

Finally, we also selected an Android application called MPG that was developed as a demonstration application for the Ford OpenXC platform in conjunction with Ford experts who are well-versed in vehicular user-interface guidelines. This application was created specifically for a vehicle touch screen by a University of Michigan undergraduate student who was not involved in the AMC project.

4.2 Comparison with industry expert

We evaluated the quality of AMC's results by comparing them with the results of a manual evaluation by an industry expert. Since the expert had limited testing time available, we took several steps to reduce the manual testing time. First, we compared only 5 of the 7 categories reported by AMC (text word count, button size, button distance, animation, and scrolling). Manually evaluating the number of user actions per task was too time-consuming because the expert would have to reason about all possible paths through the application to evaluate each action. The expert did not evaluate contrast ratio because the measurement requires special software tools that were not available at the time.

Second, the expert evaluated only 12 of the 14 applications, omitting Gmail and Google Maps because they were too complex to evaluate thoroughly given a limited time budget. Finally, rather than ask the expert to exhaustively count all violations of each property in each application, we asked the expert to render only a binary decision: either the application upholds the property or it contains one or more violations.

Prior to manual testing, we informed the expert of the specific properties that we were checking for violations. However, we did not disclose AMC's results to the expert prior to testing. After the expert completed an independent assessment, we compared results for each application. In a few cases, AMC found violations that the expert did not discover (usually due to limited testing time), but which the expert agreed were indeed violations. We used this revised expert assessment as ground truth (i.e., we suspect the expert would have found these given unlimited testing time).

Table 1 compares the results of AMC and manual expert testing. For each form of testing, a check indicates no violation, and an x indicates that the application violates the best practice guideline. A triangle in the AMC column shows that AMC could not determine

	Text word count		Button size		Button distance		Animation		Scrolling	
Popular Apps	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert
Google Finance	×	×	×	×	×	×	✓	✓	×	×
MyDays	×	×	✓	✓	×	×	✓	✓	×	×
eBay	✓	×	△	✓	△	✓	△	✓	×	×
Twitter	×	×	△	✓	×	×	✓	✓	×	×
Vehicular Apps	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert
GasBuddy	×	×	✓	✓	×	×	✓	✓	×	×
Google Navigation	✓	✓	×	×	△	×	✓	✓	✓	✓
TomTom Navigation	×	×	×	×	✓	✓	✓	✓	×	×
Waze	✓	✓	△	×	×	×	△	✓	×	×
iRadar	×	×	×	×	×	×	✓	✓	✓	✓
Beat the Traffic	✓	✓	△	✓	×	×	△	✓	✓	✓
Best Parking	✓	✓	✓	✓	×	×	✓	✓	×	×
Industry App	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert
MPG	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

This table compares AMC's results against those produced by manual testing by an industry expert. For each combination of application and usability property, the table shows AMC's result and the expert's result. A × indicates a violation, a ✓ indicates no violation, and a △ shows that AMC was not able to make a definitive judgment.

Table 1: Comparison of AMC testing with manual expert testing

on its own whether the application violated the user interface guideline. In these cases, one or more screens are flagged for subsequent manual analysis.

AMC reached a definitive judgment on 85% of the application/property combinations. Over 98% of these judgments were correct; i.e., they matched the ground truth expert assessment.

AMC had one false negative (word count for eBay). In this case, AMC was not able to visit the state that had the violation because of its state equivalence heuristic. The state is only reachable via multiple sub-menus that unfortunately all have the same structure. AMC incorrectly concludes that the sub-menu states are the same, and so it prematurely halts exploration of the sub-menu structure.

AMC cannot render a definitive judgment in 15% of the cases. Three of these cases are animations. Because this property is hard to assess, AMC defers any reasonably sized artifact for manual analysis. While the expert decided that all cases were not violations, AMC still substantially reduced the amount of manual analysis required. There are 381 total states discovered by AMC in the 12 applications. Of these, 49 states contain some form of animation, and AMC flagged only 4 of those states for manual analysis.

The remaining cases for which AMC cannot render a definitive judgment involve clickable regions in custom views. In these cases, AMC determined that view changed after clicking on a region, thereby rendering subsequent analysis useless. Again, although AMC cannot always provide a definitive judgment, it only flags a small set of states for manual analysis. There are 97 custom views in the 381 total states; AMC flagged only 13 of these for manual analysis. Across all five properties, AMC flagged less than 5% of application screens for manual analysis.

Although we did not compare results for 2 AMC properties, we believe that results for those properties will be similar to or better than the results for the other 5 properties. The violation criteria for both properties can be expressed quantitatively, and therefore AMC would not have to defer any checks for subsequent manual analysis.

4.2.1 Microbenchmarks

Table 2 shows more information about AMC's state exploration for the 14 programs. The second column shows AMC runtime. The median runtime is approximately 73 minutes, and the average runtime is slightly more than 2 hour and 33 minutes. There is clearly great variation in the time needed to explore an application, with the fastest application (Best Parking) taking about 10 minutes and the slowest (Twitter) taking a little over 10 hours.

Many factors contribute to exploration time. More complex applications that have more states and transitions naturally take longer to explore. The amount of seemingly non-deterministic behavior in an application has a large effect on exploration time. If AMC has difficulty reaching a particular target state with an unexplored transition, it may need to try many different routes before arriving at the target. Finally, applications with many custom views have longer exploration times because AMC must send many events to each custom view to discover the size and number of clickable regions within the view.

We believe AMC's exploration times are already quite reasonable since our tool executes offline, i.e., it requires no human supervision while it runs. Much of AMC's time is spent extracting the DOM tree for each state after a transition; this is a slow process on our older Nexus One phone, but it would run considerably faster on a modern phone. Additionally, exploration is potentially a highly parallelizable task. One can test multiple applications in parallel and potentially even test the same application on different computers with some minimal coordination among the test platforms.

The last three columns of Table 2 show the actual number of states in each application (which we computed by hand), the total number of states visited by AMC, and the total number of states that are visited by a purely random walk through the application. For the random walk, we configured our tool to select a random user action on each screen, wait for a transition, and then repeat. We terminated the random walk after it had been running for the exact same amount of time used by AMC for exploration. Since the random walk strategy can not easily handle custom views, we omitted checking such views for the purposes of this comparison.

Application	Total runtime	Number of states	States found by AMC	States found by random walk
Popular Apps				
Gmail	0:56:16	30	27	26
Google Finance	0:54:59	20	18	10
MyDays	1:55:03	41	26	15
eBay	0:50:29	29	24	19
Twitter	10:25:25	82	65	53
Vehicular Apps				
GasBuddy	4:28:06	67	59	50
Google Maps	1:59:26	47	31	27
Google Navigation	0:36:48	24	15	14
TomTom Navigation	4:05:13	50	36	20
Waze	6:09:15	98	81	70
iRadar	1:04:56	23	23	22
Beat the Traffic	0:19:02	18	18	16
Best Parking	0:10:26	15	11	8
Industry App				
MPG	1:22:31	5	5	4

Table 2: Time to evaluate applications and the number of states explored

For every application, AMC explores more states than the random walk strategy. However, AMC only visits 80% of the total states (as opposed to 65% for random walk). Some states are configuration-dependent: they can only be triggered by toggling application-specific menu items in the correct order. For instance, the MyDays application supports 7 different languages, but AMC explores only 2 of these languages (because of its heuristic of clicking on only 2 list items unless a significant difference in behavior is detected). It is possible, but unlikely, that the application will violate a user interface guideline in one language but not another; if so, AMC would fail to detect this.

For such applications, there exists a tradeoff between coverage and exploration time. Relaxing some of AMC's heuristics would lead to a larger number of states explored, but would likely give a poor return on the time spent in terms of finding new violations.

4.2.2 Assessment of current Android applications

After establishing that AMC is an effective tool for measuring vehicular user interface properties, we used AMC to assess the user interfaces of the 14 Android applications described in Section 4.1. Table 3 shows the total number of each type of violation detected for these applications. In the rare cases where the assessment of AMC and the industry expert differed, we use the expert's opinion after viewing the AMC results as the definitive judgment as to whether or not a violation exists.

Unsurprisingly, the results show that applications that are not designed for vehicular environments frequently violate the best practice design principles for which we tested. Since one would naturally expect the total number of violations to increase with application complexity, we normalize results by dividing the total number of violations by the number of unique application states. By this metric, the popular non-vehicular applications average 0.75 violations per state.

When we began our study, we expected the vehicular applications to more closely follow best practice guidelines. However, the 8 vehicular applications in the Android marketplace are only slightly better at following best practice guidelines and have a similar ratio of violations to total states (0.69).

The only application for which we found no violations was MPG. This application was developed with cooperation from Ford engineers to demonstrate the capabilities of the OpenXC platform. Although the primary developer was not familiar with vehicular user interface design, he likely benefited from feedback from engineers who had previously developed vehicular applications.

The best practice guidelines that were most often violated were user actions per task and text contrast ratio. This is unsurprising since these are the hardest two properties to check. Reasoning about the minimum number of actions to perform a task requires thinking about multiple states at once. Checking text contrast ratio requires special tools and/or doing complex calculations. On the other hand, we found no violations of the animation property: all instances of animation that AMC observed were judged by the industry expert to be acceptable.

Overall, these results support our conjecture that current vehicular application developers often make poor user interface design decisions. The most likely reasons for this are that they are unaware of best practices and they lack tools that can measure how their GUIs will perform in vehicular settings. AMC is designed to fill this gap.

4.3 Discussion

Since many mobile applications are supported by advertising, we had originally suspected that advertising would lead to many violations. Yet, no application that we studied had advertisements that directly triggered a violation. For instance, we did not encounter any advertisements with animation. Of course, advertisements may reduce the screen area available for application controls and indirectly lead to other violations.

AMC assumes that application behavior is mostly deterministic. If an application displays random pop-ups, for instance, AMC will only observe a violation if a pop-up occurs during exploration. Data retrieved from the cloud can also be non-deterministic. For instance, Waze displays nearby users on a map. If AMC exploration runs in the wee hours of the morning, then no users may be nearby; if exploration runs during rush hour, many users may be shown on the same map. AMC flags the map screen as an instance of animation that needs human judgment in the latter case, but not in the former case.

	Total states	User actions per task	Text contrast ratio	Text word count	Button size	Button distance	Animation	Scrolling
Popular Apps								
Gmail	30	17	2	2	0	0	0	3
Google Finance	20	0	1	3	1	4	0	3
MyDays	41	20	4	2	0	2	0	1
eBay	29	0	1	1	0	0	0	2
Twitter	82	24	28	7	0	14	0	9
Vehicular Apps								
GasBuddy	67	12	14	1	0	13	0	4
Google Maps	47	8	7	3	0	7	0	1
Google Navigation	24	0	0	0	2	1	0	0
TomTom Navigation	50	26	0	6	1	0	0	6
Waze	98	72	15	0	1	5	0	4
iRadar	23	0	2	1	5	4	0	0
Beat the Traffic	18	5	7	0	0	1	0	0
Best Parking	15	0	0	0	0	1	0	1
Industry App								
MPG	5	0	0	0	0	0	0	0

Table 3: Number of violations found for each application

AMC exploration currently only uses GUI buttons and Android built-in actions (e.g., the back button). Therefore, it does not yet find states that can only be reached by performing complex actions such as multi-touch.

5. RELATED WORK

To the best of our knowledge, AMC is the first tool to apply model checking to verify user interface properties for environments such as vehicles.

Model checking has been used to solve a diverse set of problems such as finding software bugs [4, 5, 26] and validating protocols [2, 23]. When model checking has been applied to testing application GUIs, it has been used both to discover the screens that a user would encounter and to check for faults such as GUI behavior that does not match an application specification.

Most GUI model checking systems require a pre-specified model of the GUI states and transitions [8, 17, 20]. For instance, the work of Takala et al. requires detailed definitions of GUI states and the locations of buttons within each state. Providing such a detailed model is an onerous task that requires substantial knowledge of the application's behavior.

Salvucci [18] created a cognitive model of using in-vehicle interfaces that predicts the time required by a driver to perform interactions with such interfaces. Such a model could be used to refine the heuristics (e.g., for maximum task length) that AMC uses to check for violations.

Several systems have attempted to reverse-engineer the GUI model using techniques similar to those employed by AMC. One such method is to transparently record a user's interactions with the application and generate a model based on those observations [3, 11]. The inferred model must be subsequently validated by the user. Thus, unlike AMC, which explores the GUI autonomously, these methods require a substantial amount of user time to generate the state model.

Similar to AMC's state exploration, a few methods do not require manual interaction to infer GUI states. GUI Ripping [14] infers the GUI model by automatically exploring the application in a depth-first manner. GUI Ripping defines the state by extracting all widgets, properties, and values of the window. Amalfitano et al. [1] apply a similar technique to Android applications, but their system requires application source code to detect all buttons in a state. AutoBlackTest [13] incrementally builds a GUI model as it explores by employing heuristics about what a state should be. None of these approaches deal well with applications that have both mobile and cloud components. It is challenging to checkpoint such applications, and they can exhibit the seemingly non-deterministic behavior discussed in Section 3.3.

AMC needs no external information to detect violations since the design principles it verifies are not application-specific. In contrast, most of the cited prior systems require some specification of the correct behavior of the program; for instance that a particular transition should cause the GUI to display a given state. Alternatively, systems such as AutoBlackTest generate test cases automatically, but leave it to the tester to manually verify the output of each test case. Thus, substantial manual effort is required either before or after exploration. AMC avoids this manual effort except in rare cases where it cannot make a definitive judgment.

There has been substantial effort in the usability community to determine best practices for vehicular interface design. This effort has resulted in quantitative best practice guidelines [7, 10, 16, 19, 21]. Currently, industry HMI experts use these and similar guidelines to manually verify the user interfaces of vehicular applications. AMC applies the same guidelines, but does so in an automated fashion. AMC is the first automated tool to validate vehicular interfaces.

6. CONCLUSION

Mobile computing systems are increasingly integrated with our day-to-day activities, moving beyond tablets and phones and into other devices and settings. In each new setting, different constraints on effective user interfaces will come into play. Application designers must target their products to these constraints, a difficult process even when designing an application specifically for one of these settings. Expert review can help with this process, but such review is costly and has long turnaround times, limiting its effectiveness—particularly early in the design cycle.

To reduce this pressure, we have developed a technique based on model checking that allows automatic exploration of an application's interface, verifying that particular elements of that interface meet the appropriate guidelines. This provides feedback in minutes or hours, and identifies most violations without false positives. The few remaining areas of concern can be the focus of human expert review or can be used to target the designer's efforts more efficiently, simplifying the design and verification process.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rajesh Balan, for comments that improved this paper. We also thank the vehicular HMI experts who provided advice during the design and implementation of AMC.

7. REFERENCES

[1] AMALFITANO, D., FASOLINO, A. R., AND TRAMONTANA, P. A GUI crawling-based technique for android mobile application testing. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops* (Berlin, Germany, March 2011), pp. 252–261.

[2] BLANCHET, B. Automatic verification of correspondences for security protocols. *Journal of Computer Security 17*, 4 (2009), 363–434.

[3] BROOKS, P. A., AND MEMON, A. M. Automated GUI testing guided by usage profiles. In *Proceedings of the 23rd IEEE/ACM international conference on Automated software engineering* (Atlanta, GA, November 2007), pp. 333–342.

[4] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (December 2008), pp. 209–224.

[5] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (2006).

[6] CALDWELL, B., COOPER, M., REID, L. G., AND VANDERHEIDEN, G. Web Content Accessibility Guidelines (WCAG) 2.0, 2008. `http://www.w3.org/TR/WCAG20/#contrast-ratiodef`.

[7] DRIVER FOCUS-TELEMATICS WORKING GROUP. Statement of principles, criteria and verification procedures on driver interactions with advanced in-vehicle information and communication systems. Tech. rep., Alliance of Automobile Manufacturers, June 2003.

[8] DWYER, M. B., CARR, V., AND HINES, L. Model checking graphical user interfaces using abstractions. In *Proceedings of the 6th European Software Engineering Conference* (Zurich, Switzerland, September 1997), pp. 244–261.

[9] FITTS, P. M. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology 47* (1954), 381–391.

[10] GREEN, P., LEVISON, W., PAELKE, G., AND SERAFIN, C. Suggested human factors design guidelines for driver information systems. Tech. rep., The University of Michigan Transportation Research Institute (UMTRI), August 1994.

[11] GRILO, A., PAIVA, A., AND FARIA, J. Reverse engineering of GUI models for testing. In *Proceedings of the 5th Iberian Conference on Information Systems and Technologies* (Santiago de Compostela, Spain, June 2010), pp. 1–6.

[12] Hierarchy Viewer. `http://developer.android.com/tools/help/hierarchy-viewer.html`.

[13] MARIANI, L., PEZZÈ, M., RIGANELLI, O., AND SANTORO, M. Autoblacktest: a tool for automatic black-box testing. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, May 2011), pp. 1013–1015.

[14] MEMON, A., BANERJEE, I., AND NAGARAJAN, A. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th working conference on reverse engineering* (Victoria, B.C., Canada, November 2003), pp. 260–270.

[15] MUSUVATHI, M., PARK, D. Y., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 75–88.

[16] NAKAMURA, Y. JAMA guideline for in-vehicle display systems. Tech. rep., Japan Automobile Manufacturers Association, Oct 2008.

[17] PAIVA, A. C., FARIA, J. C., TILLMANN, N., AND VIDAL, R. A. A model-to-implementation mapping tool for automated model-based GUI testing. In *Proceedings of the 7th International Conference on Formal Engineering Methods* (Manchester, United Kingdom, November 2005), pp. 450–464.

[18] SALVUCCI, D. D. Predicting the effects of in-car interfaces on driver behavior using a cognitive architecture. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, March 2001), pp. 120–127.

[19] STEVENS, A., QUIMBY, A., BOARD, A., KERSLOOT, T., AND BURNS, P. Design guidelines for safety of in-vehicle information systems. Tech. rep., Transport Research Laboratory, Feb 2002.

[20] TAKALA, T., KATARA, M., AND HARTY, J. Experiences of system-level model-based GUI testing of an android application. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation* (Berlin, Germany, March 2011), pp. 377–386.

[21] TASK FORCE HMI. European statement of principles on human machine interface for in-vehicle information and communication systems. Tech. rep., Commission of the European Communities, December 1998.

[22] Tesseract-ocr. `http://code.google.com/p/tesseract-ocr/`.

[23] TSUCHIYA, T., AND SCHIPER, A. Model checking of consensus algorithm. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems* (Beijing, China, October 2007), pp. 137 –148.

[24] UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`.

[25] VALMARI, A. A stubborn attack on state explosion. *Formal Methods in System Design 1*, 4 (1992), 297–322.

[26] YANG, J., SAR, C., AND ENGLER, D. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 131–146.

[27] ZIEFLE, M. Effects of display resolution on visual performance. *Human Factors: The Journal of the Human Factors and Ergonomics Society 40*, 4 (1998), 554–568.