Reverse Engineering Mobile Application User Interfaces With REMAUI

Tuan Anh Nguyen, Christoph Csallner
Computer Science and Engineering Department
The University of Texas at Arlington
Arlington, TX 76019, USA
Email: tanguyen@mavs.uta.edu, csallner@uta.edu

***Abstract*—When developing the user interface code of a mobile application, in practice a big gap exists between the digital conceptual drawings of graphic artists and working user interface code. Currently, programmers bridge this gap manually, by reimplementing the conceptual drawings in code, which is cumbersome and expensive. To bridge this gap, we introduce the first technique to automatically Reverse Engineer Mobile Application User Interfaces (REMAUI). On a given input bitmap REMAUI identifies user interface elements such as images, texts, containers, and lists, via computer vision and optical character recognition (OCR) techniques. In our experiments on 488 screenshots of over 100 popular third-party Android and iOS applications, REMAUI-generated user interfaces were similar to the originals, both pixel-by-pixel and in terms of their runtime user interface hierarchies. REMAUI's average overall runtime on a standard desktop computer was 9 seconds.**

I. Introduction and Motivation

Developing the user interface code of mobile applications is cumbersome and expensive in practice. Due to the early consumer and entertainment focus of the two major platforms Android and iOS and the high competitive pressure in the mobile application market, users have come to expect mobile user interfaces that are highly customized and optimized for the task at hand [42], [45]. To satisfy this demand, mobile user interfaces often deviate from their platforms' standard user interface (UI) components and provide their own novel or customized UI elements such as buttons, dividers, and custom element positioning and grouping.

To create such optimized user interfaces, the development process of mobile applications routinely incorporates non-programmers. User experience (UX) designers and graphic artists design, customize, and optimize each screen of the user interface with a mix of prototyping techniques. Common prototyping techniques include paper-and-pencil and pixel-based concept drawings created in Photoshop or similar graphic design tools [25], [17], [42], [30].

Our key observation is that there is a gap in the production process, as user interface concept drawings have to be converted into working user interface code. Currently, these conversions are done manually by programmers, which is cumbersome, error-prone, and expensive. While modern IDEs such as Eclipse, Xcode, and Android Studio have powerful interactive builders for graphical user interface (GUI) code [55], [56], using such a GUI builder to re-create a complex user interface drawing is a complex task. For example, in an evaluation of GUI builders on a set of small tasks, subjects using Apple's Xcode GUI builder introduced many bugs that later had to be corrected. Subjects produced these bugs even though the study's target layouts were much simpler than those commonly found in third-party mobile applications [56].

This challenge is compounded in practice. (1) First, custom layouts are often desired but it is harder to create them with a stock GUI builder. (2) Second, the conversion from user interface concept drawing to user interface code is typically performed many times during an application's lifespan. The reason is that many development teams follow an iterative approach, in which a user interface may undergo many revisions during both initial software development and maintenance.

This gap in the mobile application development process is significant as many mobile applications are being developed and maintained. For example, In the USA over 90% of consumers over 16 years of age use a mobile phone and more than half of the mobile phones are smartphones, mostly running Android or iOS [52]. On these smartphones, people use mobile applications to perform many tasks that have traditionally been performed on desktop computers [28], [3], [52], [23]. Example tasks include reading and writing emails, listening to music, watching movies, reading the news, and consuming and producing social media. To date, more than one million mobile applications have been released[1]. Automating the conversion from user interface design drawings to working user interface code may therefore save a lot of time and money, which could be put to better use.

Converting a conceptual drawing of a screen into good user interface code is hard, as it is essentially a reverse engineering task. As in other reverse engineering tasks, general principles have to be inferred from specific instances. For example, a suitable hierarchy of user interface elements has to be inferred from a flat set of concrete pixels.

Compared to other reverse engineering tasks such as inferring design documents from code [13], [21], [7], [15], [16], an unusual additional challenge is that the input, i.e., the pixels, may originate from scanned handwriting and human sketches with all their imperfections [53], [43], [6]. This means that sets of pixels have to be grouped together and recognized heuristically as images or text. Then groups of similar images and text have to be recognized heuristically as example elements of collections. And for the UI of innovative mobile applications, at each step the recognized elements may diverge significantly from the platform's standard UI elements.

[1] http://www.appbrain.com/stats/number-of-android-apps

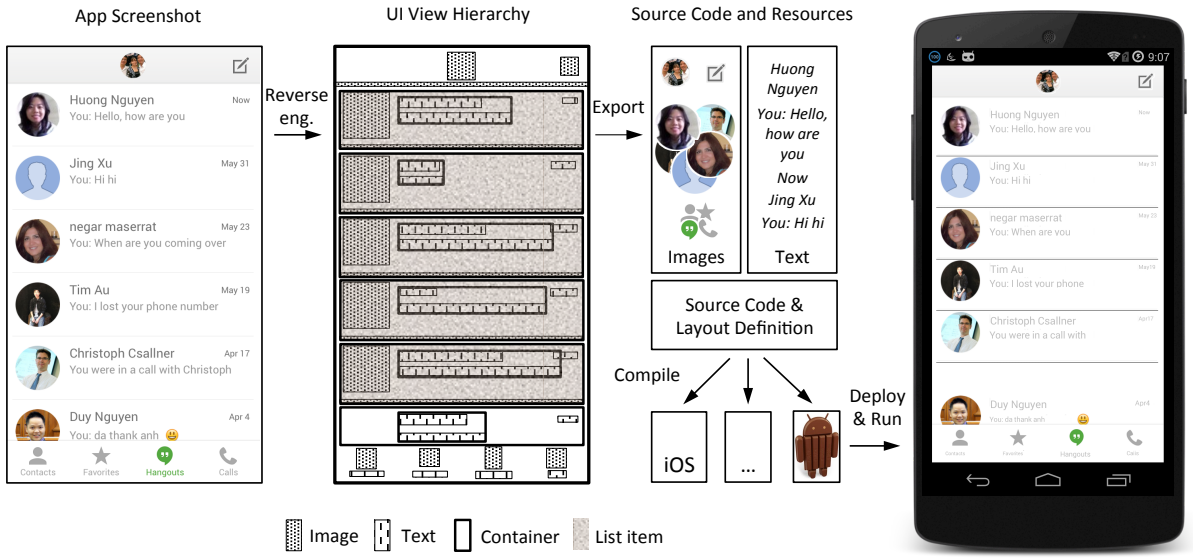

Fig. 1. Example REMAUI use: The UI designer provides a conceptual UI drawing (left). REMAUI identifies UI elements such as lists of text and images and arranges them in a suitable UI hierarchy. REMAUI then exports the inferred UI as source code and resource files, compiles them, and runs them on a phone.

For professional application development, one may wonder if this reverse engineering step is artificial. That is, why are meaning and source code hierarchy of screen elements not explicitly encoded in the conceptual design drawings if these are done in digital tools such as Photoshop? One reason is that some UX designers start with pencil on paper, so it would be desirable to convert such drawings directly into working user interface code.

More significantly, when UX designers create digital bitmap images (typically by drawing them in Photoshop), the digital design tools do not capture the hierarchy information that is needed by user interface code. More importantly, it is not clear if UX designers and graphic artists want to think in terms of source code hierarchies.

While this gap is most apparent in forward engineering, there may also exist a traditional reverse engineering scenario. A developer may only have access to screenshots of a mobile application, maybe after losing all other software artifacts such as the source code. In such a situation it would be desirable to automatically infer from the screenshots the user interface portion of the missing source code.

This paper therefore identifies and addresses three problems in mobile application development. In reverse engineering, we address the problem of inferring the user interface code of a mobile application from screenshots. In forward engineering, we address the gap between scanned pencil-on-paper UI sketches and code as well as the gap between pixel-based UI sketches and code. While these problems occur at different times in the development process, they share the task of pixels-to-code inference.

Specifically, this paper introduces the first technique to automatically *Reverse Engineer Mobile Application User Interfaces* (REMAUI). REMAUI automatically infers the user interface portion of the source code of a mobile application from screenshots or conceptual drawings of the user interface. On a given input bitmap REMAUI identifies user interface elements such as images, text, containers, and lists, via computer vision and optical character recognition (OCR) techniques. REMAUI further infers a suitable user interface hierarchy and exports the results as source code that is ready for compilation and execution. The generated user interface closely mimics the user interface of a corresponding real application. To summarize, the paper makes the following major contributions.

- The paper describes REMAUI, the first technique for inferring mobile application user interface code from screenshots or conceptual drawings.
- To evaluate REMAUI, we implemented a prototype tool that generates the UI portion of Android applications. This tool is freely available via the REMAUI web site.
- In an evaluation on 488 screenshots of over 100 popular third-party mobile applications, REMAUI-generated UIs were similar to the originals, pixel-by-pixel and in their runtime UI hierarchy.

II. Motivating Example

As a motivating example, assume a UX designer has produced the screen design bitmap shown in the left of Figure 1. The top of the screen contains the user's profile image and an icon. Below is a list, in which each entry has a person's image on the left, the person's name and text message in the middle, and the message date on the right. List entries are separated by horizontal bars. The bottom of the screen has four icons and their labels.

REMAUI infers from this bitmap working UI code, by mimicking the steps a programmer would take. REMAUI thus uses vision and character recognition techniques to reason about the screen bitmap. REMAUI groups related pixels into text or images, lines of text into text boxes, related items into containers, and repeated elements into list elements. REMAUI thus identifies non-standard user interface components such as arbitrarily shaped items (e.g., the round images on the left) and non-standard lists (e.g., using the special horizontal separator).

```
<RelativeLayout <!-- List Entry ... --> >
  <ImageView <!-- Horizontal Bar ... --> />
  <ImageView android:id="@+id/ImageView_1"
    android:layout_width="59dip"
    android:layout_height="61dip"
    android:layout_marginLeft="5dip"
    android:layout_marginTop="0dip"
    android:src="@drawable/img_9"
    android:scaleType="fitXY"
    android:layout_alignParentTop="true"
    android:layout_alignParentLeft="true"/>
  <RelativeLayout <!-- Nested: Text Block (center) ... --> >
    <TextView <!-- Sender name ... --> />
    <TextView <!-- Message ... --> />
  </RelativeLayout>
  <TextView <!-- Message date (right) ... --> />
</RelativeLayout>
```

Listing 1. REMAUI-generated layout for each list entry of Figure 1. Details are only shown for the left part of a list entry.

REMAUI generates several XML files to capture the screen's static properties. In our example, the main XML file declares and positions the elements of the top and bottom rows including icons and their labels. This file also contains a list view for the bulk of the screen content. The layout of each list entry is defined by the Listing 1 XML file. For example, it positions a contact's image and aligns it with the top left of its parent (*alignParentTop, alignParentLeft*). REMAUI recognizes aligned text blocks such as the sender's name and message, groups them into a (nested) layout container (Listing 1), and exports the recognized text fragments as an Android resource file. At application runtime the list entries are added by the also generated Listing 2 Java source code.

```
public class MainActivity extends Activity {
  //..
  private void addListView0() {
    ListView v = (ListView) findViewById(R.id.ListView_0);
    final ArrayList<ListI> values = new ArrayList<ListI>();
    values.add(new ListI(R.drawable.img_4, R.drawable.img_9, R.
        string.string_0, R.string.string_1, R.string.string_2));
    //..
  }
}
//..
```

Listing 2. REMAUI-generated Android (i.e., Java) source code that populates Listing 1 list entries at application runtime.

The generated UI code and layout definitions can be compiled with standard Android development tools. Moreover, the code is similar to how a professional developer would implement the screen. For example, the generated code uses the appropriate kinds of layout container such as RelativeLayout for the list entries. A RelativeLayout can eliminate the need for some nested containers and thus keep the layout hierarchy relatively flat, which improves rendering performance at application runtime.

III. Background

This section contains necessary background information on GUI programming, modern mobile phone GUIs, and computer vision and optical character recognition (OCR).

A. GUI View Hierarchy & Declarative GUI Programming

The graphical user interface (GUI) of many modern desktop and mobile platforms is structured as a view hierarchy [37], [2]. Such a hierarchy has two types of nodes, leaf nodes (images, buttons, text, etc.) and container nodes. The root view represents an application's entire space on screen. The root can have many transitive children. Each child typically occupies a rectangular sub-region of its parent. Each view can have its own parameters such as height, width, background color, and position. A view can be positioned relative to the root or other views such as its parent or siblings.

Mobile platforms such as Android and iOS render a parent view before its children on screen. A child view thus hides parts of its parent. Siblings are drawn in the order they are defined. A best practice is to minimize rendering time waste by keeping hierarchies flat and avoiding view overlap.

Given the relatively small mobile phone screen size, mobile platforms make it easy to hide their default screen elements such as the iOS title bar or the Android navigation bar. Applications often use this feature to maximize screen size.

To define basic GUI aspects, modern platforms provide two alternatives. The traditional desktop approach is construction through regular program code [37]. The now widely recommended alternative is declarative [2], [39], [24], e.g., via XML layout definition files in Android. Advanced GUI aspects are then defined programmatically, which typically leads to a combination of code and layout declaration files.

Building an appealing user interface is hard [36], [37]. Besides understanding user needs, the GUI facilities of modern platforms are complex and offer many similar concepts to choose from. This challenge is especially significant for developers new to their target platform. While each platform provides standard documentation and sample code, these samples often produce unappealing results.

B. Example GUI Framework: Android

The Android standard libraries define various GUI containers ("layout containers") and leaf nodes ("widgets"). According to an August 2012 survey of the 400 most popular non-game applications in the Google Play app store [47], the following containers were used most frequently: LinearLayout (130 uses per application on average) places its children in a single row or column; RelativeLayout (47) positions children relative to itself or each other; FrameLayout (15) typically has a single child; ScrollView (9) is a scrollable FrameLayout; and ListView (7) lays out children as a vertical scrollable list.

The following widgets were used most frequently: TextView (141) is read-only text; ImageView (62) is a bitmap; Button (37) is a device-specific text button; View (17) is a generic view; EditText (12) is editable text; and ImageButton (11) is a device-independent button that shows an image. Besides the above, the Android library documentation currently lists some additional two dozen widgets and some three dozen layout containers.

C. Optical Character Recognition (OCR)

To infer UI code that closely reproduces the input conceptual drawing, REMAUI distinguishes text from images and

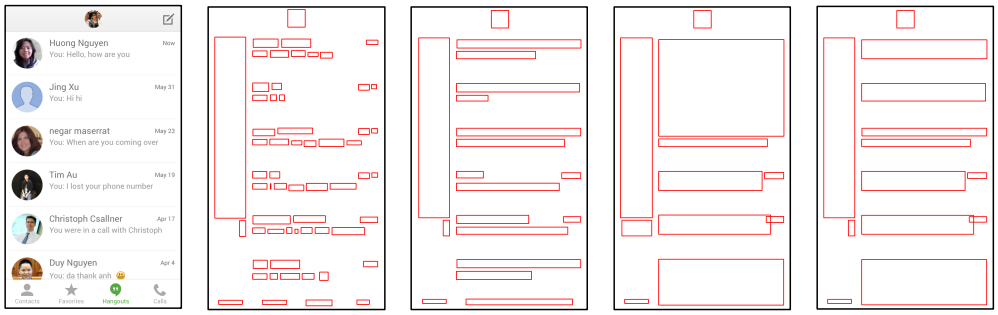

Fig. 2. Example OCR performance at various granularity levels. Left to right: UI drawing and Tesseract-detected words, lines, blocks, and paragraphs.

captures the text as precisely as possible. Decades of research into optical character recognition (OCR) have produced specialized methods for recognizing various kinds of text such as text in different sizes, fonts, and orientation, as well as handwritten text [53], [43]. Generally it is easier to recognize text online (while it is being written) than offline. Similarly, it is easier to recognize print than handwriting.

Existing OCR tools perform relatively well if the input consists of mostly text. A good example is single-column text with few images. Current OCR tools perform worse if the text density is lower and text is arranged more freely and combined with images [27]. A good representative OCR tool is the powerful and widely used open-source OCR engine Tesseract [48], [50], which, for instance, Mathematica 9 uses to recognize text. In the closely related task of segmenting pages (for example, to distinguish images and individual text columns), Tesseract performs on par with commercial tools [50], [1].

However, the limitations of such a powerful OCR tool on complex inputs become apparent when subjecting it to screenshots or conceptual UI drawings. For example, Figure 2 shows from left to right a conceptual drawing and Tesseract's results when detecting text at various granularity levels, i.e., words, lines, blocks, and paragraphs. In this example Tesseract found all words but also classified as words non-words such as the contacts' images. In general, for the domain of conceptual screen drawings and screenshots Tesseract's precision and recall are often both below one in all granularity levels. So even a powerful OCR tool may miss some words and classify non-text as words.

IV. REMAUI Overview and Design

Figure 3 shows REMAUI's six main processing steps. At the core is a powerful off-the-shelf optical character recognition (OCR) engine (step 1). Since OCR produces false positive candidate words, REMAUI filters the OCR results with its domain-specific heuristics. Both to further compensate for OCR's limitations and to identify non-text elements such as images, REMAUI combines OCR with a powerful off-the-shelf computer vision system (step 2).

In two-dimensional images computer vision techniques can quickly detect features such as corners and edges. Computer vision has therefore been applied to diverse tasks such as recognizing faces in two-dimensional images of the world or to allow self-driving cars to detect the edge of the road [51]. Using computer vision REMAUI approximates the boundaries of each screen element such as text and images.

In its final steps REMAUI merges OCR and computer vision results (step 3) and in the merged data identifies structures such as lists (step 4). REMAUI then exports the inferred user interface as a combination of layout declarations and program source code for the given target mobile platform (step 5), compiles this combination to binaries, and runs the binaries on an unmodified smartphone (step 6).

Not shown in Figure 3 is a pre-processing step in which REMAUI removes standard operating system title and navigation bars, if they are present. Since these screen areas are standardized it is relatively easy to detect and remove them.

A. Optical Character Recognition (Step 1)

First, REMAUI applies on the given input bitmap off-the-shelf OCR word detection. Since optical character recognition suffers from false positives, REMAUI post-processes OCR results to remove candidate words that likely do not reflect true words in the input. Figure 4 visualizes this process on the example bitmap from Figure 3. At word-level detection, REMAUI's OCR system classifies several UI elements as a word that are not a word but an image or a part of an image.

To remove likely false positive words, REMAUI encodes knowledge about its mobile phone UI domain as heuristics, summarized in Table I. As an example, rule 3 encodes that on a phone screen a word is likely not cut off and thus does not extend beyond the border of the screen. This rule is specific to phone screens and does not apply in all the settings the off-the-shelf OCR engine may be applied in outside REMAUI.

TABLE I. Heuristics for eliminating likely false positive candidate words from the OCR results.

#	Name	Heuristic
1	Zero	$h = 0 \vee w = 0$
2	Long	$w/h < 0.05 \vee h/w < 0.05$
3	Cut off	$x < 0 \vee y < 0 \vee x + w > W \vee y + h > H$
4	Conf.	$c \leq 0.4$
5	Content	$c \leq 0.7 \wedge (\frac{\lvert e_h/e_w - h/w \rvert}{max(e_h/e_w, h/w)} > 0.5 \vee \frac{\lvert a-e \rvert}{max(a,e)} > 0.8)$
6	No-text	`[\p{C}\s]*` ∨ `[^\\x00-\\x7F]*`

The heuristics are given in terms of the input data, the OCR results, and heuristic values computed by REMAUI. Specifically, from the input UI screen available are its width (W) and height (H). The OCR system produces for each of its candidate words the word's height (h), width (w), area ($a = w * h$), font family and size, upper left corner coordinates (x, y), text

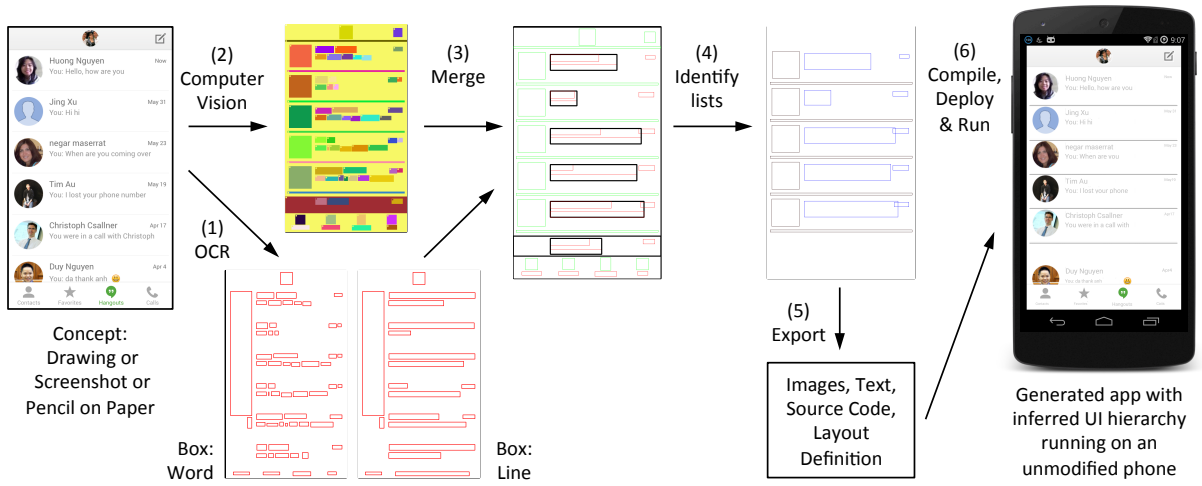

Fig. 3. Overview of REMAUI processing steps: (1) Locate and extract candidate words and lines with OCR; (2) locate and extract candidate UI elements as a hierarchy of nested bounding boxes using computer vision; (3) merge the results to improve recognition quality; (4) identify repeated items and summarize them as collections; (5) export the constructed UI as a mobile application for a given platform; (6) compile and execute.

content (t), and confidence level (c). The confidence level is derived from the distance of the word's characters from idealized characters [49].

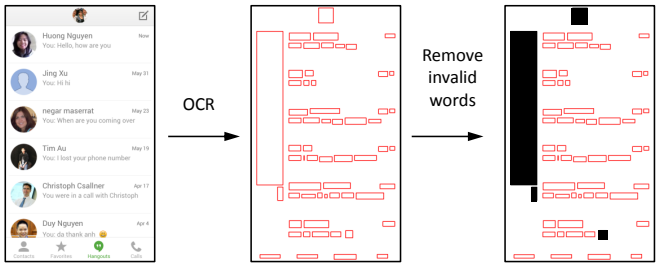

Fig. 4. Example results of the Table I heuristics: Input from Figure 3 (left), candidate words from OCR (framed, middle), and candidates eliminated by our heuristics (solid rectangles, right).

From the text content and font information produced by OCR for a given word, REMAUI estimates the width (e_w), height (e_h), and area (e) the candidate word should occupy given the font size and family. Rule 5 uses this information to remove a word if, within bounds, the text area estimated by REMAUI does not match the text area reported by OCR. This rule removed all four candidate words that are removed in the right side of Figure 4.

The other rules exclude words OCR is not confident about (rule 4), have a zero dimension (rule 1), or have an odd shape (rule 2). An odd shape likely does not capture an English-language word, as they are long and narrow, vertically or horizontally. Finally, rule 6 removes words that only contain non-ASCII characters or only consist of control characters and whitespace.

The heuristics' constants are derived through trial and error on a small set of third-party bitmaps. The resulting heuristics have held up reasonably well on the much larger set of third-party bitmaps used in the evaluation (Section VI).

B. *Computer Vision (Step 2)*

In this step REMAUI infers a first candidate view hierarchy. Two important observations are that (1) many vastly different view hierarchies can lead to very similar if not identical on-screen appearances and (2) a programmer will likely find some of these view hierarchies more valuable than others. REMAUI therefore follows carefully chosen heuristics to produce desirable view hierarchies that balance the following two goals.

The first goal is a minimal hierarchy, i.e., having a minimum number of nodes. From the programmer's perspective this is important to prevent clutter in the generated code. More importantly, drawing a large number of views slows down the application. For example, a programmer would not want a container that contains one child view for each character of every word displayed by the container.

However, a competing goal is maximum flexibility of the inferred view hierarchy. Distinct UI elements should be represented by distinct views to allow the generated UI to be well displayed on various combinations of screen size and resolution. Thus, a programmer would, for instance, not want to represent the four distinct buttons of the Figure 3 bottom-screen navigation bar as a single image. However, combining these four buttons into a single image and a single leaf view would reduce the number of views.

To infer a good candidate view hierarchy, REMAUI first tries to identify all atomic visual elements in the input UI. By atomic we mean a visual element that reasonably should not be divided further. For example, an icon is atomic but so can be an entire text paragraph. For each identified atomic visual element REMAUI then computes its approximate view.

To achieve these tasks, REMAUI leverages off-the-shelf computer vision. Figure 5 illustrates REMAUI's key computer vision steps on the Figure 3 example input bitmap. First we detect the edges of each image element via Canny's widely used algorithm [10], [51]. But these edges themselves are not good candidates for atomic elements as, for example, each character or even minor noise would become its own element.

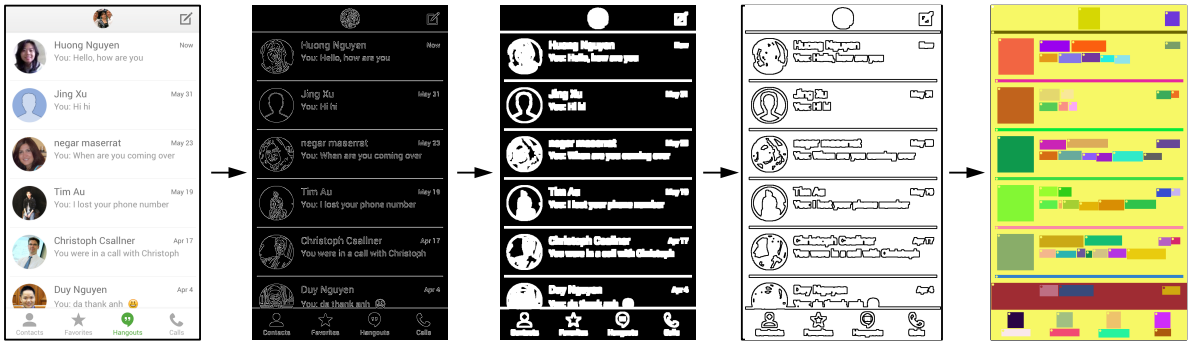

Fig. 5. Computer vision processing steps from left to right: Original input bitmap; Edges detected via Canny's algorithm as black and white; Dilated or broadened edges to swallow noise and join adjacent elements; Contours of the joined elements; Output: Hierarchy of the contours' bounding boxes.

To merge close-by elements with each other and with surrounding noise and to close almost-closed contours REMAUI dilates its detected edges. REMAUI uses a heuristic to, for example, allow a word's characters to merge but keep words separate. REMAUI then computes the dilated edges' contours. Each contour is a candidate atomic element.

Figure 5 also illustrates the heuristic nature of this process. The last list entry shown in the input screen is cut off by a horizontal dividing line. Edge detection, dilation, and contour thus all merge the last list item with the dividing line, reducing REMAUI's precision and recall of atomic visual elements.

Finally, REMAUI computes the bounding box of each candidate atomic element, to approximate the element's view. Recall from Section III-A that typically each view is rectangular and fully contained in its parent. Partially overlapping boxes are thus merged into a new bounding box. A fully contained box becomes the child view of the containing box.

C. Merging (Step 3)

In two sub-steps REMAUI merges the results of OCR and computer vision to heuristically combine the best aspects of both and to integrate the OCR-inferred text into the vision-inferred candidate view hierarchy.

First, REMAUI removes OCR-detected words that conflict with vision-inferred element bounding boxes. This step addresses common OCR false positives such as classifying part of an image as a text fragment, classifying bullet points as "o" or a similar character, and merging lines of text that have too little spacing. The resulting OCR-extracted text is not useful and should instead be exported as an image.

Specifically, each OCR word is subjected to the Table II heuristics. In addition to the OCR word's width (w) and height (h), we now also have the computer vision bounding box's width (b_w) and height (b_h). For example, rule (1) checks if an OCR word overlaps with two vision boxes whose y-coordinates do not overlap. This happens if OCR merged two text lines whereas the vision results kept them separate.

REMAUI further removes OCR words that are not contained by an OCR line (using the OCR lines from step 1). REMAUI then merges OCR words and lines into text blocks.

TABLE II. HEURISTICS FOR ADDITIONAL ELIMINATIONS OF OCR WORDS, BASED ON COMPUTER VISION RESULTS.

#	Description
1	Word aligns vertically & overlapped $\geq$ 70% with $\geq$ 2 vision boxes that do not overlap each other
2	Word aligns horizontally & overlapped $\geq$ 70% with $\geq$ 2 vision boxes, distance between each pair of boxes $>$ each box's size
3	Word contains a non-leaf vision box
4	Word contains only 1 vision box, box size $<$ 0.2 word size
5	Non-overlapped leaf vision box contains only 1 word, word size $<$ 0.2 box size
6	If leaf vision box's words are $>$ 50% invalidated, invalidate the rest
7	If $>$ 3 words are the same text and size, aligned left, right, top, or bottom, each has $<$ 0.9 confidence, and are non-dictionary words
8	Leaf vision box contains a word, $M < 0.4 \vee (M < 0.7 \wedge m < 0.4) \vee (M >= 0.7 \wedge m < 0.2)$, with $m = min(\frac{w}{b_w}, \frac{h}{b_h}), M = max(\frac{w}{b_w}, \frac{h}{b_h})$

OCR lines often blend together into a single line unrelated text that just happened to be printed on the same line. For example, the Figure 3 contact names (left) appear on the same line as message dates (right). However they are conceptually separate. REMAUI thus splits a line if the word-level OCR indicates that the distance between two words exceeds a heuristic threshold (i.e., their height). Figure 6 shows this process for the Figure 3 example. REMAUI adds the resulting text blocks to the view hierarchy and removes the vision boxes they overlap with.

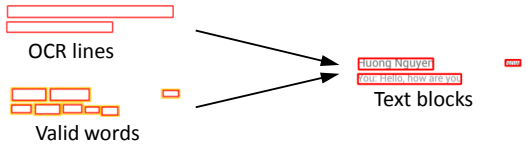

Fig. 6. Example: Merging Figure 3 OCR lines with processed OCR words.

REMAUI aims at extracting text contents with high precision. The employed OCR engine produces better text contents when treating its input as a single text line. This way the OCR engine does not have to reason about which parts of the input are text versus non-text. REMAUI thus invokes OCR on each text block (in line mode), yielding text that resembles the text in the input relatively closely. Finally, REMAUI groups close-by text blocks into a container, if the vertical distance between text blocks is less than either of their heights.

D. Identify Lists (Step 4)

In this step REMAUI identifies repeated items and summarizes them as collections, for two reasons. First, the final UI definition is more compact and efficient if each repeated resource is only represented once. Second, this step allows REMAUI to generalize from a few instances to a generic collection. REMAUI can then supply the observed instances as an example instantiation of the collection.

REMAUI identifies repeated instances by ordering the views by their relative location and searching them for identical sub-trees. A sub-tree consists of a view and a subset of its transitive children. Two sub-trees are identical if each of their child nodes has a peer in the other sub-tree, such that both nodes have the same number of children and the same width, height, type (text or image), and matching location within its parent (each within a threshold). Neither text contents nor image bitmaps have to be identical, as a list item may, for example, contain the face of a user as in Figure 3.

If identical sub-trees are found, REMAUI creates a bounding box around each of them. Each box contained in such a bounding box that is not part of the sub-tree belongs to the list item anchored by the sub-tree. However, such an overlapping box varies across list elements and will be exported as an optional element of the list entry. The properties of these optional elements are determined by overlaying all of them and using the resulting bounding boxes.

E. Export (Step 5)

In this step REMAUI exports all results as an Android project directory, complete with relevant source code and resource files. This directory can be compiled with standard Android IDEs. Specifically, REMAUI crops and extracts each identified image from the input screenshot, only once for repeated images. To provide a reasonable background color, REMAUI uses as the container's background the dominant color of each container after extracting all identified images. REMAUI exports all detected text content and format to Android *strings.xml* and *styles.xml* files. REMAUI exports layout files to the Android *layout* directory, for the layout shared between list entries and for the main screen. Finally, REMAUI generates Java code to fill lists with the identified entries at runtime.

V. Research Questions

To evaluate REMAUI, we ask (a) if it is currently feasible to integrate REMAUI into a standard mobile application development setup and (b) if REMAUI-generated user interfaces are useful in the sense that the generated UI is similar to the UI an expert developer would produce. We therefore investigate the following three research questions (RQ), expectations (E), and hypotheses (H).

- RQ1: What is REMAUI's runtime in a standard development setup?
  - E1: Given its expensive OCR and computer vision techniques, we do not expect REMAUI to run interactively.
  - H1: REMAUI can run on a standard development machine in a similar amount of time as a software installation wizard, which we approximate as up to one minute.
- RQ2: Is a REMAUI-generated UI visually (pixel by pixel) similar to a given third-party input UI conceptual drawing?
  - E2: Given their wide variety, we do not expect REMAUI to work well for all applications.
  - H2: REMAUI produces a UI that is visually similar to an input UI conceptual drawing, when running on non-game mobile applications.
- RQ3: Is the view hierarchy of a REMAUI-generated UI similar to the view hierarchy of a given third-party input application?
  - E3: Given their wide variety, we do not expect REMAUI to work well for all applications.
  - H3: REMAUI produces a UI whose view hierarchy is similar to the view hierarchy of a given handwritten non-game mobile application.

VI. Evaluation

To explore our research questions we implemented REMAUI for Android. Our prototype generates Android code and resource files that are ready to be compiled and executed. Our prototype supports, among others, Android's three most popular layout containers and three most popular widgets (Section III-B). For off-the-shelf OCR, REMAUI uses the open source engine Tesseract [48] and Tesseract's default version 3.0.2 English language data trained model. This means that REMAUI currently does not use Tesseract's options for training its classifiers. Step 1 uses Tesseract's fastest mode[2] with fully automatic page segmentation. For off-the-shelf computer vision, REMAUI uses the open source engine OpenCV [6] in its default configuration, without training.

A REMAUI-generated application's aspect ratio (between output screen width and height) is the same as the one of its input screenshot. With this aspect ratio, a REMAUI-generated application supports many screen resolutions, via Android's standard density-independent pixel (dp) scheme. The Android runtime thereby scales a REMAUI-generated application's dp units based on a device's actual screen density.

The high-level workflow of our experiment is as follows. We first ran a subject Android or iOS application on a corresponding phone. At some point we took a screenshot and at the same time captured the current UI hierarchy. We (only) handed the captured screenshot to REMAUI and thus obtained a generated application. We then ran the generated application on an Android phone and, at the same time, took a screenshot and captured the runtime hierarchy. To clarify, no UI hierarchy information was provided to REMAUI.

Obtaining the UI hierarchy at runtime required low-level OS access. We thus used a rooted Google Nexus 5 phone for Android (2 GB RAM, Android 4.4.4) and a jail-broken iPhone 5 (1 GB RAM, iOS 7.1.2). To obtain the view hierarchy on Android we used Android's uiautomator[3] via the Android Debug Bridge [41] (*adb shell uiautomator dump*). For iOS, we used cycript [26] recursively starting from the root view.

[2] TESSERACT_ONLY, PSM_AUTO

[3] http://developer.android.com/tools/help/uiautomator

For the evaluation REMAUI ran on a 16 GB RAM 2.6 GHz Core i7 MacBook Pro running OS X 10.10.2.

A. Subjects

Using existing third-party applications to explore our research questions is a good fit for several reasons. (1) First, it is straightforward to capture a screenshot of a running application and hand such a screenshot to REMAUI. It is also straightforward to compare such screenshots pixel by pixel with REMAUI-generated screenshots (RQ2). (2) More importantly, having a running application enables inspecting the application's UI hierarchy. We can then compare this hierarchy with the corresponding UI hierarchy of the REMAUI-generated application (RQ3).

Since our REMAUI prototype is implemented for Android, our first group of subjects consists of third-party non-game Android applications. To sample popular applications, we downloaded the top-100 free Android applications from the Google Play store as of November 9, 2014. From these we excluded games, as most games do not provide GUIs through a view hierarchy but through the native OpenGL library. This left us with 46 top-100 applications, covering (except games) all application categories present in the top-100, such as e-commerce, email, maps, media players, productivity tools, translation software, and social media. The REMAUI web site lists name and version of each subject application used in the evaluation. From each application, we captured the application's main screen (in the form it appears after starting the application). We refer to these subjects as group C.

To broaden our set of subjects, and since many developers first target iOS, we added iOS applications. We downloaded on August 12, 2014 the top 100 free iOS applications from the Apple App Store. The resulting 66 non-game top-100 applications cover a range of categories similar to group C. We took a screenshot of every screen we could reach, yielding 302 screenshots (group A). For each application we took another screenshot showing the main screen with different data contents, yielding 66 subjects (group B). Since iOS 7 defined a new design language and Google and Apple are major application developers, we included from them 58 screenshots of iOS 7 applications outside the top-100 (group D).

There may also be a use case of manually drawing designs and scanning them. Since such third-party drawings are hard to obtain, we created sketches of 16 screenshots (group E). These screenshots are our manual renderings of the main screen of the alphabetically first 16 of the top 100 iOS applications in the Apple app store as of August 12, 2014.

B. RQ1: REMAUI Runtime

Figure 7 shows the runtime of REMAUI's seven major processing steps, i.e., (1) OCR, (2) computer vision, (3a) merging OCR text with vision boxes, (3b) splitting text lines, (3c) creating the view hierarchy, (4) identifying lists, (5) export, and total runtime. Each step shows the runtimes of groups A–E from left to right. Not surprisingly, steps 1, 2, and 3b took longest, as these are the only steps that call REMAUI's computer vision and OCR engines. The cost of step 3b varied widely, as the number of OCR calls depends on the number of identified text blocks. Step 5 includes extracting a bitmap for each image view, which also takes time. On a modern desktop computer total runtime was well within the one minute time frame, with a 52 second maximum and an average total runtime of 9 seconds.

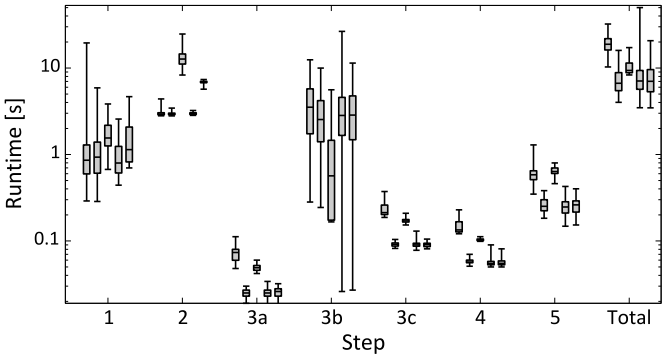

Fig. 7. Runtime of REMAUI's seven main processing steps on the 488 subjects, shown by group, from left (A) to right (E).

C. RQ2: Pixel-by-Pixel Similarity

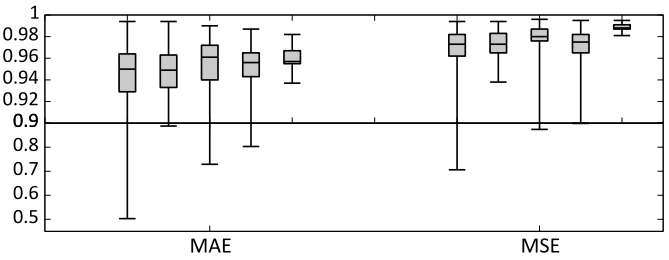

Fig. 8. Normalized pixel-by-pixel screenshot similarity between REMAUI input and generated application on the 488 subjects, shown by group A–E from left to right. Higher values are better.

Since REMAUI currently removes all standard OS status and navigation bars from input screenshots, we do the same to the screenshots of REMAUI-generated applications. To ensure that input and generated screenshots have the same dimensions, we set the target application screen dimensions to account for subtracting the OS navigation bar.

We used the open source Photohawk[4] library to measure two widely used picture similarity metrics [51]. Specifically, following are the mean absolute error (MAE) and the mean squared error (MSE) over a screenshot's n pixels; $e_{i,j}$ is the delta of one of the three color channels RGB of a given pixel in the original vs. the corresponding pixel in the REMAUI-generated screenshot.

$$\text{MAE} = \frac{1}{3n}\sum_{i=1}^{n}\sum_{j=1}^{3}|e_{i,j}| \qquad \text{MSE} = \frac{1}{3n}\sum_{i=1}^{n}\sum_{j=1}^{3}e_{i,j}^2$$

Figure 8 shows the normalized (to $[0, 1]$) similarity measures for our 488 subjects, arranged by our five groups. The results indicate that REMAUI-generated applications achieved high average pixel-by-pixel similarity with the respective inputs on both metrics.

[4]http://datascience.github.io/photohawk

D. RQ3: UI Hierarchy Similarity

Achieving high pixel-by-pixel similarity is not sufficient, as it is trivially achieved by a UI that consists of a single bitmap (i.e., the input screenshot). So in this section we also evaluate how closely the generated view hierarchy resembles the view hierarchy that produced REMAUI's input screenshot.

Evaluating the quality of a given UI hierarchy is hard. First, there are often several different hierarchies of similar overall quality. Not having the application's specification, it is not clear which alternative REMAUI should target. Second, unlike for RQ2, the input application's UI hierarchy is often not an obvious gold standard. Many of the subjects contained redundant intermediate containers that have the same dimension as their immediate parent and do not seem to serve a clear purpose. Other container hierarchies could have been refactored into fewer layers to speed up rendering.

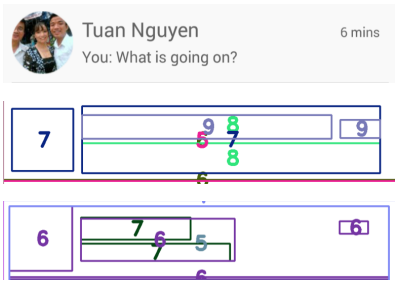

Fig. 9. Part of a screenshot of Google Hangout (top), its UI hierarchy (middle), and the REMAUI-generated hierarchy (bottom). Each element is annotated at its center with its level in the UI hierarchy, with root=1. Each number's color matches the color of its element's boundary.

Figure 9 shows an example of this challenge. The original UI hierarchy and the REMAUI-generated one differ in several aspects. For example, REMAUI puts the contact's name and message into two relatively small level-7 text boxes. The original application puts the same strings into much larger level-8 text boxes. Similarly, REMAUI groups name and message into a level-6 container. The original application groups them with the date into a level-7 container. This container is nested into a level-6 container, which is nested into a level-5 container of the same dimensions. The latter container thus seems redundant. Despite these differences, screenshots of the two hierarchies are very similar pixel-by-pixel.

In our evaluation we side-step these challenges by comparing UI hierarchies at the leaf level. While this comparison does not capture the entire hierarchy, it still captures parts of the UI's structure. For example, the boundary of each intermediate (container) node is represented by the leaf nodes it contains.

Specifically, for this experiment we analyzed each pixel in a REMAUI-generated screenshot. If a pixel belongs to a text box in both the original and the generated application, then we consider the pixel correct. Similarly, the pixel is correct if it belongs to an image view in both the original and in the generated application. Given these criteria, we can define precision p and recall r as follows, separately for images, text, and overall, given the pixels i in an image view in the original application and in the generated application (i') as well as the pixels t in a text view in the original application and in the generated application (t').

$$p_i = \frac{|i \cap i'|}{|i'|} \quad p_t = \frac{|t \cap t'|}{|t'|} \quad r_i = \frac{|i \cap i'|}{|i|} \quad r_t = \frac{|t \cap t'|}{|t|}$$

$$p = \frac{|i \cap i'| + |t \cap t'|}{|i'| + |t'|} \qquad r = \frac{|i \cap i'| + |t \cap t'|}{|i| + |t|}$$

Since in our setup this experiment required manual steps for capturing the UI hierarchies we restricted the scope of the experiment to our core group of Android subjects (group C) and the relatively small group of iOS subjects (group B). Figure 10 shows the experiment's results. We found a moderate to high structural match (in terms of the leaf nodes) between original and REMAUI-generated UI hierarchies.

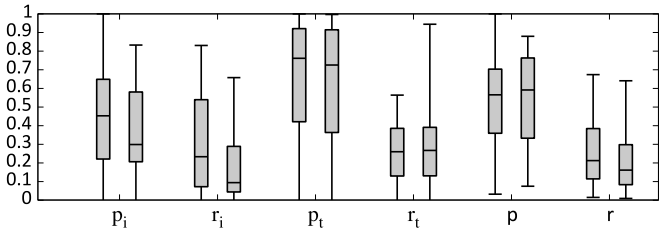

Fig. 10. Image, text, and overall UI element precision (p) and recall (r) for groups B (left) and C (right). Higher values are better.

The low recall in Figure 10 does not fully capture how REMAUI reproduced text or images. On the contrary, REMAUI's pixel-by-pixel similarity was high (Figure 8). We suspect a culprit of low recall was white-space. REMAUI computes tight bounding boxes, but a corresponding original text or image view may contain much additional white-space and fill its parent container (as the much larger text boxes in Figure 9).

To explore this issue on the example of text, we measured the Levenshtein distance (edit distance) [33] of text box strings between original and generated applications. For the 2.4k string pairs of group C, on average, an original text was 18.8 characters, a generated text 14.6, and the edit distance 4.7. So on average it took only 4.7 single-character additions, removals, or substitutions to convert a string in the generated application back to the corresponding string in the original application. For the 2.9k group B string pairs, on average, an original text was 14.6 characters, a generated text 15.0, and the edit distance was only 2.9. These results indicate a higher text recall than the pixel-based recall of Figure 10.

The following two trends emerged on manual inspection. First, precision suffered if a subject contains a bitmap that contained both text and non-text imagery. This is not surprising, as for OCR it is hard to distinguish if a given text is plain text or belongs to a bitmap of text and other elements. REMAUI typically decomposed such bitmaps into text and image views. The resulting UI hierarchy should be relatively easy to fix manually. A developer would just replace a generated container (containing both text and images) with a single bitmap. Overall these incorrectly detected views were small. In Figure 10, their average area was less than 0.25% of the input screen area.

The second observation is that low image recall occurred when images overlapped that were of similar color or where

the top image is somewhat transparent. These scenarios are challenging for edge detection. Similarly, text recall was low if the text color was similar to the background color. On the flip-side, with high contrast we observed high recall.

VII. Related Work

The gap between early prototyping and formal layout definition also exists in the related domain of web site development. A study of 11 designers at 5 companies showed that all designers started with sketching the layout, hierarchy, and flow of web pages with pencil on paper and in graphical design tools such as Photoshop [40].

A similar design process has been reported for desktop applications. At Apple, user interface sketches were first created with a fat marker (to prevent premature focus on details) and later scanned [54]. Separate studies of hundreds of professionals involved in UI design in various companies indicated heavy use of paper-based sketches [31], [9]. One of the reasons was that sketching on paper is familiar due to designers' graphic design background.

Despite much progress in tools for creating user interfaces that combine the unique talents of graphic designers and programmers [38], [12], much conceptual user interface design work is still being done by graphic designers with pencil on paper and digitally, e.g., in Photoshop. Previous work has produced fundamentally different approaches to inferring user interface code, as it was based on different assumptions. Following are the main changed assumptions for mobile application UI development and reverse engineering that motivate our work. (1) First, many UX designers and graphic artists do not construct their conceptual drawings using a predefined visual language we could parse [32], [8], [14], [46]. (2) Second, while this was largely true for desktop development, mobile application screens are not only composed of the platform's standard UI framework widgets [19], [20]. (3) Finally, we cannot apply runtime inspection [11], [35] as REMAUI runs early in the development cycle.

Specifically, the closest related work is MobiDev [46], which recognizes instances of a predefined visual language of standard UI elements. For example, a crossed-out box is recognized as a text box. But unlike REMAUI, MobiDev does not integrate well with a professional mobile application development process. It would require UX designers and graphic artists to change the style of their paper and pencil prototypes, for example, to replace real text with crossed-out boxes. Such changes may reduce the utility of the prototypes for other tasks such as eliciting feedback from project stakeholders. In a traditional reverse engineering setting, MobiDev cannot convert screenshots into UI code.

SILK and similar systems bridge the gap between pen-based GUI sketching and programming of desktop-based GUIs [32], [8], [14]. Designers use a mouse or stylus to sketch directly in the tool, which recognizes certain stroke gestures as UI elements. But these tools do not integrate well with current professional development processes as they do not work on paper-on-pencil scans or screenshots. These tools also do not recognize handwritten text or arbitrary shapes.

UI reverse engineering techniques such as Prefab [19] depend on a predefined model of UI components. The work assumes that the pixels that make up a particular widget are typically identical across applications. However, this is not true for a mobile application UI. Mobile applications often have their own unique, non-standard identity, style, and theme. For Prefab to work, all possible widget styles and themes of millions of current and future mobile applications would need to be modeled.

PAX [11] heavily relies on the system accessibility API at program runtime. At runtime PAX queries the accessibility API to determine the location of text boxes. The accessibility API also gives PAX the text contents of the text box. PAX then applies computer vision techniques to determine the location of words in the text. If a view does not provide accessibility, PAX falls back to a basic template matching approach. PAX thus cannot reverse engineer the UI structure of mobile applications from their screenshots or application design images alone.

Recent work applies the ideas of SILK and DENIM to mobile applications [18], allowing the user to take a picture of a paper-and-pencil prototype. The tool allows the user to place arbitrary rectangles on the scanned image and connect them with interaction events. This idea is also implemented by commercial applications such as Pop for iOS. As SILK and DENIM, this approach is orthogonal to REMAUI.

VIII. Conclusions and Future Work

When developing the UI code of a mobile application, a big gap exists between graphic artists' conceptual drawings and working UI code. Programmers typically bridge this gap manually, by reimplementing the conceptual drawings in code, which is cumbersome and expensive. To bridge this gap, we introduced the first technique to automatically Reverse Engineer Mobile Application User Interfaces (REMAUI). On a given input bitmap REMAUI identifies UI elements via computer vision and OCR techniques. In our experiments on 488 screenshots of over 100 popular third-party applications, REMAUI-generated UIs were similar to the originals, both pixel-by-pixel and in terms of their runtime UI hierarchies.

We plan to (1) generalize the export step to additional platforms such as iOS and cross-platform JavaScript-based frameworks. (2) REMAUI currently converts each input screen to a separate application. We plan to provide a graphical notation to allow users to connect several input screens drawings, which REMAUI could use to generate a single application with various screens and corresponding screen transitions. (3) We plan to integrate REMAUI with tools that generate mobile application functionality either via keyword-based code search [44] or from high-level models [34], [5], [22] or DSLs [29], [4]. (4) We plan to index a screenshot corpus by running REMAUI on it and storing REMAUI's intermediate results. Exposing this index via a query interface would allow a user to search for screenshots by their structure and features.

The REMAUI prototype for Android used in the evaluation is freely available at: http://cseweb.uta.edu/~tuan/REMAUI/

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1117369.

REFERENCES

[1] A. Antonacopoulos, S. Pletschacher, D. Bridson, and C. Papadopoulos, "ICDAR 2009 page segmentation competition," in *Proc. 10th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, Jul. 2009, pp. 1370–1374.

[2] Apple Inc., "View programming guide for iOS," https://developer.apple.com/library/ios/documentation/windowsviews/conceptual/viewpg_iphoneos/ViewPG_iPhoneOS.pdf, Oct. 2013, accessed May 2015.

[3] P. Bao, J. S. Pierce, S. Whittaker, and S. Zhai, "Smart phone use by non-mobile business users," in *Proc. 13th Conference on Human-Computer Interaction with Mobile Devices and Services (Mobile HCI)*. ACM, Aug. 2011, pp. 445–454.

[4] S. Barnett, R. Vasa, and J. Grundy, "Bootstrapping mobile app development," in *Proc. 37th ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, May 2015.

[5] M. Book and V. Gruhn, "Modeling web-based dialog flows for automatic dialog control," in *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, Sep. 2004, pp. 100–109.

[6] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, 1st ed. O'Reilly, Oct. 2008.

[7] Y. Brun and M. D. Ernst, "Finding latent code errors via machine learning over program executions," in *Proc. 26th ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, May 2004, pp. 480–490.

[8] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge, "JavaSketchIt: Issues in sketching the look of user interfaces," in *Proc. AAAI Spring Symposium on Sketch Understanding*. AAAI, Mar. 2002, pp. 9–14.

[9] P. F. Campos and N. J. Nunes, "Practitioner tools and workstyles for user-interface design." *IEEE Software*, vol. 24, no. 1, pp. 73–80, Jan. 2007.

[10] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, Nov. 1986.

[11] T.-H. Chang, T. Yeh, and R. C. Miller, "Associating the visual representation of user interfaces with their internal structures and metadata," in *Proc. 24th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, Oct. 2011, pp. 245–256.

[12] S. Chatty, S. Sire, J.-L. Vinot, P. Lecoanet, A. Lemort, and C. P. Mertz, "Revisiting visual interface programming: creating GUI tools for designers and programmers," in *Proc. 17th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, Oct. 2004, pp. 267–276.

[13] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan. 1990.

[14] A. Coyette, S. Kieffer, and J. Vanderdonckt, "Multi-fidelity prototyping of user interfaces." in *Proc. 11th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT)*. Springer, Sep. 2007, pp. 150–164.

[15] C. Csallner and Y. Smaragdakis, "Dynamically discovering likely interface invariants," in *Proc. 28th ACM/IEEE International Conference on Software Engineering (ICSE), Emerging Results Track*. ACM, May 2006, pp. 861–864.

[16] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, May 2008, pp. 281–290.

[17] T. S. da Silva, A. Martin, F. Maurer, and M. S. Silveira, "User-centered design and agile methods: A systematic review," in *Proc. Agile Conference (AGILE)*. IEEE, Aug. 2011, pp. 77–86.

[18] M. de Sà, L. Carriço, L. Duarte, and T. Reis, "A mixed-fidelity prototyping tool for mobile devices," in *Proc. Working Conference on Advanced Visual Interfaces (AVI)*. ACM, May 2008, pp. 225–232.

[19] M. Dixon and J. Fogarty, "Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure," in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Apr. 2010, pp. 1525–1534.

[20] M. Dixon, D. Leventhal, and J. Fogarty, "Content and hierarchy in pixel-based methods for reverse engineering interface structure," in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, May 2011, pp. 969–978.

[21] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[22] J. Falb, T. Röck, and E. Arnautovic, "Using communicative acts in interaction design specifications for automated synthesis of user interfaces," in *Proc. 21st ACM/IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 2006, pp. 261–264.

[23] L. Fortunati and S. Taipale, "The advanced use of mobile phones in five European countries," *The British Journal of Sociology*, vol. 65, no. 2, pp. 317–337, Jun. 2014.

[24] M. Gargenta and M. Nakamura, *Learning Android: Develop Mobile Apps Using Java and Eclipse*, 2nd ed. O'Reilly, Jan. 2014.

[25] Z. Hussain, M. Lechner, H. Milchrahm, S. Shahzad, W. Slany, M. Umgeher, T. Vlk, and P. Wolkerstorfer, "User interface design for a mobile multimedia application: An iterative approach," in *Proc. 1st International Conference on Advances in Computer-Human Interaction (ACHI)*. IEEE, Feb. 2008, pp. 189–194.

[26] Jay Freeman, "Cycript," http://www.cycript.org/, 2014, accessed May 2015.

[27] D. Karatzas, F. Shafait, S. Uchida, M. Iwamura, L. G. i Bigorda, S. R. Mestre, J. Mas, D. F. Mota, J. Almazàn, and L. de las Heras, "ICDAR 2013 robust reading competition," in *Proc. 12th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, Aug. 2013, pp. 1484–1493.

[28] A. K. Karlson, B. Meyers, A. Jacobs, P. Johns, and S. K. Kane, "Working overtime: Patterns of smartphone and PC usage in the day of an information worker," in *Proc. 7th International Conference on Pervasive Computing (Pervasive)*. Springer, May 2009, pp. 398–405.

[29] A. Khambati, J. C. Grundy, J. Warren, and J. G. Hosking, "Model-driven development of mobile personal health care applications," in *Proc. 23rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, Sep. 2008, pp. 467–470.

[30] K. Kuusinen and T. Mikkonen, "Designing user experience for mobile apps: Long-term product owner perspective," in *Proc. 20th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2013, pp. 535–540.

[31] J. A. Landay and B. A. Myers, "Interactive sketching for the early stages of user interface design," in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, May 1995, pp. 43–50.

[32] ——, "Sketching interfaces: Toward more human interface design," *IEEE Computer*, vol. 34, no. 3, pp. 56–64, Mar. 2001.

[33] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, Feb. 1966.

[34] A. Martínez, H. Estrada, J. Sánchez, and O. Pastor, "From early requirements to user interface prototyping: A methodological approach," in *Proc. 17th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, Sep. 2002, pp. 257–260.

[35] X. Meng, S. Zhao, Y. Huang, Z. Zhang, J. Eagan, and R. Subramanian, "WADE: simplified GUI add-on development for third-party software," in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Apr. 2014, pp. 2221–2230.

[36] B. A. Myers, "Challenges of HCI design and implementation," *Interactions*, vol. 1, no. 1, pp. 73–83, Jan. 1994.

[37] ——, "Graphical user interface programming," in *Computer Science Handbook*, 2nd ed., A. B. Tucker, Ed. CRC Press, May 2012.

[38] B. A. Myers, S. E. Hudson, and R. F. Pausch, "Past, present, and future of user interface software tools," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 7, no. 1, pp. 3–28, Mar. 2000.

[39] V. Nahavandipoor, *iOS 7 Programming Cookbook*, 1st ed. O'Reilly, Nov. 2013.

[40] M. W. Newman and J. A. Landay, "Sitemaps, storyboards, and specifications: A sketch of Web site design practice as manifested through

artifacts," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-99-1062, 1999.

[41] T. A. Nguyen, C. Csallner, and N. Tillmann, "GROPG: A graphical on-phone debugger," in *Proc. 35th ACM/IEEE International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER) track*. IEEE, May 2013, pp. 1189–1192.

[42] G. Nudelman, *Android Design Patterns: Interaction Design Solutions for Developers*. Wiley, Mar. 2013.

[43] R. Plamondon and S. Srihari, "Online and off-line handwriting recognition: A comprehensive survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 63–84, Jan. 2000.

[44] S. P. Reiss, "Seeking the user interface," in *Proc. 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, Sep. 2014, pp. 103–114.

[45] S. E. Salamati Taba, I. Keivanloo, Y. Zou, J. Ng, and T. Ng, "An exploratory study on the relation between user interface complexity and the perceived quality of Android applications," in *Proc. 14th International Conference on Web Engineering (ICWE)*. Springer, Jul. 2014.

[46] J. Seifert, B. Pfleging, E. del Carmen Valderrama Bahamóndez, M. Hermes, E. Rukzio, and A. Schmidt, "Mobidev: A tool for creating apps on mobile phones," in *Proc. 13th Conference on Human-Computer Interaction with Mobile Devices and Services (Mobile HCI)*. ACM, Aug. 2011, pp. 109–112.

[47] A. S. Shirazi, N. Henze, A. Schmidt, R. Goldberg, B. Schmidt, and H. Schmauder, "Insights into layout patterns of mobile user interfaces by an automatic analysis of Android apps," in *Proc. ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS)*. ACM, Jun. 2013, pp. 275–284.

[48] R. Smith, "An overview of the Tesseract OCR engine," in *Proc. 9th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, Sep. 2007, pp. 629–633.

[49] ——, "An overview of the tesseract ocr engine," in *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 629–633. [Online]. Available: http://www.google.de/research/pubs/archive/33418.pdf

[50] R. W. Smith, "Hybrid page layout analysis via tab-stop detection." in *Proc. 10th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, Jul. 2009, pp. 241–245.

[51] R. Szeliski, *Computer Vision: Algorithms and Applications*. Springer, Nov. 2010.

[52] The Nielsen Company, "The mobile consumer: A global snapshot," http://www.nielsen.com/us/en/insights/reports/2013/mobile-consumer-report-february-2013.html, Feb. 2013.

[53] Ø. D. Trier, A. K. Jain, and T. Taxt, "Feature extraction methods for character recognition—a survey," *Pattern Recognition*, vol. 29, no. 4, pp. 641–662, Apr. 1996.

[54] Y. Y. Wong, "Rough and ready prototypes: Lessons from graphic design," in *Proc. ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), Posters and Short Talks*. ACM, 1992, pp. 83–84.

[55] C. Zeidler, C. Lutteroth, W. Stürzlinger, and G. Weber, "The Auckland layout editor: An improved GUI layout specification process," in *Proc. 26th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, Oct. 2013, pp. 343–352.

[56] ——, "Evaluating direct manipulation operations for constraint-based layout," in *Proc. 14th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT)*. Springer, Sep. 2013, pp. 513–529.