Push-Button Synthesis of Watch Companions for Android Apps

Cong Li
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
congli@smail.nju.edu.cn

Yanyan Jiang
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
changxu@nju.edu.cn

ABSTRACT

Most Android apps lack their counterparts on convenient smartwatch devices, possibly due to non-trivial engineering efforts required in the new app design and code development. Inspired by the observation that widgets on a smartphone can be mirrored to a smartwatch, this paper presents the Jigsaw framework to greatly alleviate such engineering efforts. Particularly, Jigsaw enables a push-button development of smartphone's companion watch apps by leveraging the programming by example paradigm, version space algebra, and constraint solving. Our experiments on 16 popular open-source apps validated the effectiveness of our synthesis algorithm, as well as their practical usefulness in synthesizing usable watch companions.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Application specific development environments.*

KEYWORDS

Android apps, WearOS apps, program synthesis

ACM Reference Format:
Cong Li, Yanyan Jiang, and Chang Xu. 2022. Push-Button Synthesis of Watch Companions for Android Apps. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510056

1 INTRODUCTION

Smartwatch is a fast-growing market since Apple Watch's first release in 2015 [73]. It has been predicted that over 140 million smartwatches will be shipped in 2021 [65], and the market share of WearOS (an Android variant designed for smartwatches) will increase by 72.8% in 2022 [64]. Being lightweight and convenient, smartwatches significantly reduce user's efforts for performing simple yet regular daily tasks, e.g., receiving notifications, making a phone call, and adding a quick note [25, 37]. WearOS also attracts developers' attention: Every year, Google releases news and tutorials on the development of WearOS apps on Android Dev Summit [27]; and Twitter, Spotify, Outlook, and Google-family app all have official smartwatch counterparts on Google Play Store.

However, there is an easily overlooked gap between the smartphone developers and smartwatch (hereafter, watch) apps, making the prospect of watch apps left far behind the smartphone. Specifically, watch apps speak a considerably different design language from smartphone apps [25, 28] due to limited screen space, performance, and battery life. Porting an app to the watch platform requires non-trivial engineering efforts [15], including redesigning user experiences and adaptation to watch APIs. Consequently, there are only ~4,000 WearOS apps in Google Play Store [16], while Android apps are ~2.8 million. For a majority of apps, watch users cannot exploit the convenience that a watch brings.

As a first exploratory work to bridge the gap, we observe that an Android app's watch counterpart could be automatically synthesized by mirroring smartphone widgets to a watch. In other words, a watch app can be effectively regarded as a smartphone app's *companion* which acts as a remote agent:

(1) When the on-watch companion app is active, it is always synchronized with its associated smartphone app (running in background). The synchronization automatically migrates a subset of widgets on the Android app to the watch and renders them in a watch-friendly GUI layout.
(2) Watch users can interact with the migrated widgets. All actions (e.g., clicks) performed on the watch are delegated back to the background app on the smartphone, whose GUI changes are kept synchronized with the watch companion.

This bridging can significantly reduce the engineering efforts required for developing watch apps. For this goal, this paper presents a framework called Jigsaw for assisting developers in creating such smartphone companion apps as easily as push-button. Jigsaw leverages the programming by example [30, 32] paradigm in which developers *annotate* positive and negative widget examples (widgets should/not be mirrored) over GUI layout dumps. Then, Jigsaw automates all the rest by:

(1) synthesizing a widget selector s, a small piece of domain-specific language program (more precisely, a disjunction of XPaths for matching paths on an XML tree), that generalizes the developer's annotations. s takes any GUI layout tree as input and produces (selects) a subset of widgets that should be mirrored on the watch screen.
(2) synthesizing an installable watch companion app (by instantiating a template) for communications with the smartphone and displaying mirrored widgets.
(3) injecting a background server stub to the smartphone app for monitoring GUI changes, applying the synthesized selector

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510056

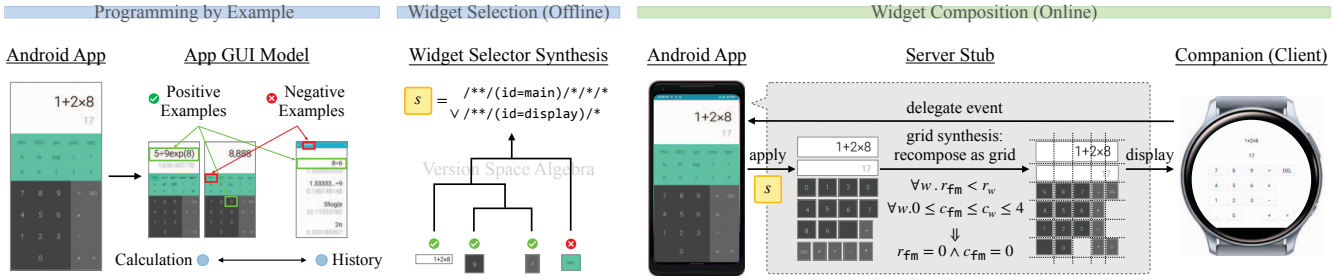

Figure 1: Jigsaw overview. With developer-annotated examples on the GUI model, Jigsaw for each GUI model state synthesizes a widget selector. When the smartphone app's GUI layout changes at runtime, its associated widget selector is activated to select widgets to mirror, reorganize them as a grid, and send them to the watch client for display. All user actions performed on the watch companion are delegated to the app.

s, and solving a grid system for appropriately positioning selected widgets such that the relative widget positions on the smartphone can also be preserved on the watch.

We implemented the Jigsaw SDK (synthesizer, app server stub, and watch client template) for automating the development of smartwatch companion apps. We evaluated Jigsaw by creating watch companions for 16 popular open-source Android apps (all without a watch counterpart yet). The results are encouraging that synthesized widget selectors well generalize annotated examples with high precision and recall, and selected widgets are useful in helping users complete real-world user tasks.

In summary, this paper's major contribution is a first exploratory report on the possibility of automating the process of synthesizing an Android app's smartwatch companion. We also open-source our prototype tool and other supplementary materials to facilitate future research in this domain:

https://sites.google.com/view/jigsaw-wapp.

The rest of the paper is organized as follows. We give an overview of Jigsaw and explain our methodology in Section 2. The Jigsaw approach and implementation details of Jigsaw SDK are presented in Section 3. We describe the evaluation in Section 4 and elaborate on related work in Section 5. This paper is concluded in Section 6.

2 METHODOLOGY

The Jigsaw workflow is shown in Figure 1. Given annotated examples, Jigsaw produces a server stub (including the synthesized widget selector) for the smartphone app and an installable WearOS APK. In this procedure, we identified two key problems[1]:

(1) **Widget selection**: how to determine which widgets should be selected to mirror on the watch?
(2) **Widget composition**: how to (re)compose selected widgets to fit the watch screen?

The challenges and our methodology to mitigate them are explained below.

[1]Other issues are mainly related to engineering.

2.1 Widget Selection: Programming by Example Modulo App States

An Android app may contain hundreds of GUIs and thousands of widgets, but the developer may like to provide only a limited number of annotated examples. It is thus not practical to synthesize a single widget selector that can generalize to all GUIs. This brings our first challenge:

Challenge 1. *How to find a mechanism to effectively synthesize widget selectors from only a few examples?*

The first step towards effective widget selector synthesis is to leverage the app's GUI model to *separately conduct program synthesis for each app state*, yielding smaller-scale program synthesis problem instances. Specifically, an app's GUI model is a finite state machine in which homogeneous GUI layouts share the same app state[2]. It is thereby natural to synthesize for each app state a separate selector because a single, concise widget selector hardly generalizes well to drastically different GUI layouts. Furthermore, app states without examples can be assigned with a trivial widget selector that selects no widget. Such absence of examples usually indicates that this app state may not have to be synchronized with the watch companion.

For example, a developer (or a dynamic analysis [4, 14, 40, 45, 70, 74]) may provide Google Calculator [58] a GUI model consisting of two states (Figure 1): Calculation and History. Widget selector synthesis is then conducted per state.

The simplified program synthesis sub-problems make the powerful tool *version space algebra* [42] a tractable choice for widget selector synthesis[3]. Particularly, we designed an automaton-based domain-specific language (DSL) for describing a disjunction of XPaths [72] denoted by directed graphs in which each edge is associated with a widget selection constraint. Applying the selector to any GUI layout tree yields a set of selected widgets. The synthesis procedure roughly works as follows. For each positive example, we construct its version space consisting of all possible selectors that are compatible with this positive example. Then, we conduct

[2]Such models can be defined by developers or automatically extracted based on the `Activitys`/`Fragments` or execution traces [4, 14, 40, 45, 70, 74].
[3]Assuming that each app state corresponds to a different widget selector, the size of version space intersections grows exponentially with the number of examples.

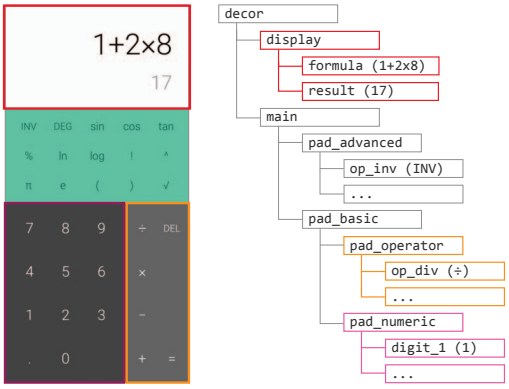

Figure 2: GUI layout for Google Calculator

intersection (generalization) of version spaces until no further generalization can be performed. Disjunction of the XPaths in the remainder version spaces yields candidate widget selectors. We filter out invalid selectors (that select negative examples) and rank the candidate for a balanced simplicity and generalizability.

For the motivating example of Google Calculator [58] in Figure 1, the synthesized selector for Figure 1's Calculation state is

```
/**/(id=main)/*/*/*
∨ /**/(id=display)/*,
```

where the first line selects the numeric and operator widgets and the second line selects the formula and calculation result widgets. Figure 2 displays the GUI layout tree. JIGSAW synthesizes two independent XPaths because numeric and operator widgets share the `main` container, and the formula and calculation result share the `display` container. We shall explain more about how a widget selector works and is synthesized later in Section 3.

2.2 Widget Composition: Solving a Grid System

The GUI layout on the phone is synchronized with the watch by invoking the synthesized widget selector whenever the app's GUI layout changes. Then, selected widgets should be positioned on the watch in a developer-friendly way. This yields our second challenge:

Challenge 2. *How to efficiently assign bound boxes for on-watch widgets to maximally preserve their relative positions at runtime?*

Our widget composition follows the *grid system*, a classical and widely used paradigm in graphic and GUI design [10, 17, 18, 36, 54]. Specifically, a grid system is an $m \times n$ matrix in which cells are containers of widgets (a widget is allowed to occupy multiple adjacent cells). Given n (the number of columns) by the developer, JIGSAW synthesizes a minimum $m \times n$ grid such that for any pair of selected (phone) widgets $w_1 \prec w_2$ ($\prec$ denotes the relative relation of bound boxes, i.e., w_1 is to the left/top of w_2), $w_1 \prec w_2$ is preserved in the synthesized grid. We intentionally made all these constraints linear, such that constraint solving for practical GUIs can be done in milliseconds.

For the motivating example in Figure 1, JIGSAW created constraints to enforce the formula widget to be placed above all other widgets, yielding our synthesized layout.

3 PUSH-BUTTON SYNTHESIS OF WATCH COMPANIONS FOR ANDROID APPS

3.1 Widget Selection: Selector Synthesis

Widgets and GUI layout. An Android app's GUI snapshot is a rooted tree (Figure 2) in which nodes are *widgets* and edges denote the direct containing relation. In a GUI layout tree, branch nodes are usually container widgets holding other widgets as children, while leaf nodes display information and handle user events. This paper defines a widget as a set of attribute-value pairs where each pair denotes an attribute and its value.

Widget selection by examples. A widget is localized by its ancestors in the GUI layout tree. Therefore, this paper defines the *widget path* for a widget w as the widget list in the shortest path from the root container w_1 to $w_n = w$ in the GUI layout tree: $\omega = [w_1, w_2, \cdots, w_n]$, where w_{i-1} is the direct parent of w_i for all $1 < i \leq n$.

JIGSAW takes developer or end-user specified positive and negative example sets E^+ and E^- (elements of E^+ and E^- are widget paths) as input[4] and synthesizes a *widget selector*, a piece of domain-specific language (DSL) program s that accepts all examples in E^+ and rejects any example in E^-, i.e.,

$$(E^+ \subseteq [\![s]\!]) \wedge (E^- \cap [\![s]\!] = \varnothing)$$

in which $[\![s]\!]$ denotes all widget paths s can accept.

Widget selectors. The syntax of widget selector s is defined as a disjunction of one or more XPaths x:

Selector	s	::=	$x \mid s \vee x$
XPath	x	::=	$/\ell \mid x/\ell$
Level Condition	ℓ	::=	$(\varphi) \mid * \mid **$
Condition	φ	::=	$attr\ op\ val \mid \varphi \wedge \varphi$
Attribute	$attr$	::=	widget attribute, e.g., `id`
Operator	op	::=	allowed operator, e.g., =
Attribute Value	val	::=	attribute value

The semantics of s ($[\![s]\!]$, all widget paths s can accept) is defined by $[\![s \vee x]\!] = [\![s]\!] \cup [\![x]\!]$, where $[\![x]\!]$ denotes all possible widget paths x can accept. Specifically, given a widget path $\omega = [w_1, w_2, \cdots, w_n]$, an XPath x matches ω ($\omega \in [\![x]\!]$) if ω can be written as a concatenation of $\omega = \omega_1 :: \omega_2 :: \cdots :: \omega_m$ such that the k-th level condition ℓ_k in $x = /\ell_1/\ell_2/\cdots/\ell_m$ matches ω_k via the following rules:

(1) "*attr op val*" matches exactly one widget only if the widget's attribute *attr* satisfies the *attr op val* condition.
(2) "$*$" matches exactly one widget no matter its attributes. "$*$" can be regarded as an always-true *op*.
(3) "$**$" matches zero or more widgets. There is an additional requirement that x cannot end with "$**$".

For example,

$$x = /*/(\texttt{id} = \texttt{'formula'} \wedge \texttt{text}\ startswith\ \texttt{'1'})$$

accepts all 2-widget paths $[w_1, w_2]$ if w_2's `id` is `formula` and `text` starts with `'1'`. Another example is $x = /**/*$ for accepting all widget paths. Since our definition of x is a subset of the XPath standard [72], we do not further formalize them.

[4]Obtaining examples are easy even for an end-user: one can provide a GUI tool for letting the users mark for the positive/negative examples.

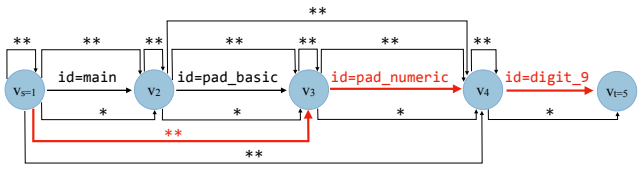

Figure 3: Graph representation of version space for widget path ω = [`decor`, `main`, `pad_basic`, `pad_numeric`, `digit_9`]. Red path derives XPath `/**/id=pad_numeric/id=digit_9`.

Version spaces. The synthesis algorithm is based on the version space algebra [42], in which each positive example

$$\omega = [w_1, w_2, \cdots, w_n] \in E^+$$

is associated with its version space $\texttt{VS}(\omega) = G(V, E)$, a labeled directed graph (with self-loops and parallel edges) for a concise representation of all XPaths that can match ω. In $G(V, E)$, each vertex $v_i \in V = \{v_1, \ldots, v_n\}$ corresponds to $w_i \in \omega$ and

$$\begin{aligned} E = &\{(v_i, v_j, **) \mid 1 \le i \le j < n\} \cup \\ &\{(v_i, v_{i+1}, *) \mid 1 \le i < n\} \cup \\ &\{(v_i, v_{i+1}, \ell) \mid \ell \in \texttt{LCond}(w_{i+1}) \wedge 1 \le i < n\} \end{aligned}$$

where $\texttt{LCond}(w)$ is all possible level conditions that can match w. Specifically, we found that the equality operator (=) suffices for widget companion synthesis in practice[5], i.e.,

$$\begin{aligned} \texttt{LCond}(w) &= \{(\textstyle\bigwedge_{\varphi \in S} \varphi) \mid S \subseteq \texttt{Cond}(w)\} \\ \texttt{Cond}(w) &= \{attr = val \mid (attr, val) \in w\}. \end{aligned}$$

Given $\texttt{VS}(\omega) = G(V, E)$, there exists exactly one "source" vertex $v_s \in V$ without incoming edge (not considering self-loops) and one "sink" vertex $v_t \in V$ of zero out-degree. Each $v_s \to v_t$ path corresponds to an XPath by mapping each label on the edge as a level condition ℓ. Furthermore, G is acyclic if self-loops are removed. Therefore, we can define $\mathcal{X}(G)$, all XPaths derived from G, as the XPaths for all $v_s \to v_t$ paths (without consecutive $**$ edges) in G. Figure 3 displays the version space of a digit widget 9 in Google Calculator.

Version spaces intersections. Intersecting two version spaces $G(V, E)$ and $G'(V', E')$ yields the graph representation of XPaths that are simultaneously acceptable to G and G'. The intersection $G \sqcap G'$ is also a graph $G_\sqcap(V_\sqcap, E_\sqcap)$ where

$$\begin{aligned} V_\sqcap &= \{\langle v, v' \rangle \mid v \in V \wedge v' \in V'\} \\ E_\sqcap &= \{(\langle u, u' \rangle, \langle v, v' \rangle, \ell \sqcap \ell') \mid (u, v, \ell) \in E \wedge (u', v', \ell') \in E'\} \end{aligned}$$

In terms that ℓ matches v' and ℓ' matches v, the intersection of level conditions $(\ell \sqcap \ell')$ is defined by the conjunction of ℓ and ℓ':

$$\begin{aligned} (\varphi) \sqcap (\varphi') &= (\varphi \wedge \varphi'), & (\varphi) \sqcap * &= (\varphi), \\ * \sqcap * &= *, & (\varphi) \sqcap ** &= \bot, \\ ** \sqcap ** &= **, & * \sqcap ** &= \bot. \end{aligned}$$

It is easy to verify that there is exactly one source vertex and one sink vertex in $V_\sqcap$, and thus $\mathcal{X}(G_\sqcap)$ is well-defined. Furthermore, our intersection soundly captures the widgets paths acceptable to XPaths in both G and G', i.e., for all $x \in \mathcal{X}(G)$ and $x' \in \mathcal{X}(G')$ that $[\![x]\!] \cap [\![x']\!] \neq \emptyset$, there exists $x_\sqcap \in \mathcal{X}(G \sqcap G')$ such that

$$[\![x]\!] \cap [\![x']\!] = [\![x_\sqcap]\!].$$

Readers may refer to supplementary material for a brief proof.

[5]Version space algebra supports other operators, e.g., string *startswith* or comparisons with constants [30]. However, these operators have only limited merits in selecting widgets.

Algorithm 1: Widget selector synthesis

```
1 Function SynthesizeSelector(E+, E−)
2     Ω ← {VS(ω) | ω ∈ E+};
3     while ∃G1, G2 ∈ Ω. ¬trivial(G1 ⊓ G2) do
4         G1*, G2* = argmax_{G1,G2 ∈ Ω, ¬trivial(G1 ⊓ G2)}
                |{G ∈ Ω | ¬trivial(G1 ⊓ G2 ⊓ G)
                    ∨ (trivial(G1 ⊓ G) ∧ trivial(G2 ⊓ G))}|
                + |X(G1 ⊓ G2)| / max{|X(G1)|, |X(G2)|};
5         Ω ← (Ω \ {G1*, G2*}) ∪ {G1* ⊓ G2*};
6     return {s = (x1 ∨ ··· ∨ x_|Ω|) | xi ∈ X(Gi), Gi ∈ Ω,
              E+ ⊆ [[s]], E− ∩ [[s]] = ∅};
```

Widget selector synthesis. Synthesizing widget selectors is conceptually simple by intersecting all version spaces

$$G(V, E) = \bigsqcap_{\omega \in E^+} \texttt{VS}(\omega).$$

Any XPath $x \in \mathcal{X}(G)$ that is consistent with positive/negative examples is a feasible selector. However, the synthesis procedure should be carefully handled in practice because:

(1) The intersection of two graphs of $O(n)$ vertices can have $O(n^2)$ vertices. This is an exponential growth with the number of examples.
(2) Intersecting version spaces may yield useless, trivial G of whom all derived XPaths are equivalent to `/**/*`, i.e.,

$$trivial(G) = \top \Leftrightarrow \forall x \in \mathcal{X}(G). [\![x]\!] = [\![\texttt{/**/*}]\!].$$

This suggests that widgets should be separately selected with disjunction ($\vee$).

Therefore, our synthesis algorithm (Algorithm 1) heuristically intersects version spaces following existing literature [30]. Specifically, we maintain a pool of version spaces Ω, which initially contains each $\omega \in E^+$'s version space (Line 2). In each iteration (Lines 3–5), we greedily intersect two version spaces that potentially yield a minimal yet non-trivial intersection (Line 5). Among non-trivial intersections, the preference score of (G_1, G_2) is calculated by letting all $G \in \Omega$ to vote (Line 4). Particularly,

(1) G votes for (G_1, G_2) if $\neg trivial(G_1 \sqcap G_2 \sqcap G)$, indicating that $G_1 \sqcap G_2$ makes it possible to continue intersecting with G in next iterations.
(2) G votes for (G_1, G_2) if $trivial(G_1 \sqcap G) \wedge trivial(G_2 \sqcap G)$, indicating that $G_1 \sqcap G_2$ does not affect afterward intersections between G and other version spaces.
(3) If multiple G share the same number of votes, we additionally consider the number of XPaths in the intersection (the last term, which is always ≤ 1).

The intersection is repeated until all further intersections are trivial. Finally, the set of all widget selectors that comply with the examples (Line 6) are returned as candidates.

Ranking candidate widget selectors. All candidates $s \in S$ are feasible selectors that are consistent with E^+ and E^-. Among them, we should choose a best selector that has the potential to generalize to unseen GUI layouts. Specifically, let $\mathcal{W}$ be all widget paths in the example GUI layout trees (either positive or negative), we prefer selectors that are structurally simple following the Occam's razor [9] while can simultaneously select as many widget paths in the examples as possible, i.e., maximizing $|[\![s]\!] \cap \mathcal{W}|$ for better generalizability. Therefore, we choose the best candidate

$$s^\star = \arg\min_{s \in S} \left(\frac{\texttt{complexity}(s)}{\max_{s' \in S} \texttt{complexity}(s')} - \frac{\left|[\![s]\!] \cap \mathcal{W}\right|}{\max_{s' \in S} \left|[\![s']\!] \cap \mathcal{W}\right|} \right)$$

as our synthesized widget selector, in which the $\texttt{complexity}(s = x_1 \vee x_2 \cdots \vee x_n) = \sum_{1 \le i \le n} \texttt{complexity}(x_i)$, and $\texttt{complexity}(x)$ for XPath x is based on frequent patterns and anti-patterns of level conditions in XPaths. Each pattern is a single level condition (e.g., $*$), a consecutive level conditions (e.g., $*/*$, $**/(\varphi)$), or a combination (e.g., $**/\cdots/*$) associated with a positive/negative score. Readers may refer to the supplementary material for all patterns.

3.2 Widget Composition: Grid Synthesis

Assigning widget coordinates. Given a set of selected widgets W, we assign each widget $w \in W$ a cell position (r_w, c_w) in the $m \times n$ grid, where n is specified by the developer and m is minimized by the constraint solver. To preserve spatial locality, we produce each pair of widgets $w, w' \in W$ a linear constraint $\phi_{w,w'}$ that if w is to the top/left of w' on the phone, w should be positioned in an upper/left cell of w' on the watch. Specifically,

$$\phi_{w,w'} = row(w, w') \wedge col(w, w')$$

$$row(w, w') = \begin{cases} 0 \le r_w < r_{w'} < m, & \text{if } b(w) \le t(w') \\ 0 \le r_w \le r_{w'} < m, & \text{if } t(w) \le t(w') < b(w) < t(w') \\ 0 \le r_w = r_{w'} < m, & \text{if } t(w) = t(w') \wedge b(w) = b(w') \\ \texttt{true}, & \text{otherwise} \end{cases}$$

$$col(w, w') = \begin{cases} 0 \le c_w < c_{w'} < n, & \text{if } r(w) \le l(w') \\ 0 \le c_w \le c_{w'} < n, & \text{if } l(w) \le l(w') < r(w) < l(w') \\ 0 \le c_w = c_{w'} < n, & \text{if } l(w) = l(w') \wedge r(w) = r(w') \\ \texttt{true}, & \text{otherwise} \end{cases}$$

where functions t, b, l, and r denote the top, bottom, left, and right bound boxes of a widget, respectively. For example, the following constraints are produced for 0 and 9:

$$\phi_{\texttt{digit\_0},\texttt{digit\_9}} = \texttt{true} \wedge 0 \le c_{\texttt{digit\_0}} < c_{\texttt{digit\_9}} \le n$$
$$\phi_{\texttt{digit\_9},\texttt{digit\_0}} = 0 \le r_{\texttt{digit\_9}} < r_{\texttt{digit\_0}} < m \wedge \texttt{true}.$$

We solve the conjunction of all above constraints

$$\Phi_W = \bigwedge_{w,w' \in W} \phi_{w,w'}$$

to obtain each widget's grid position (r, c) with a minimized m. In the example,

$$(r_{\texttt{digit\_9}}, c_{\texttt{digit\_9}}) = (2, 2), (r_{\texttt{digit\_0}}, c_{\texttt{digit\_0}}) = (5, 0).$$

Reflowing widgets. Sometimes, constraint solving may fail if the specified grid is too narrow (too small n). For example, selecting k widgets in a row requires $n \ge k$ to preserve their relative positions. In such a case, we find the minimum n' to yield a SAT and *reflow* the $m \times n'$ grid to n columns by wrapping the overflowed columns like the rendering of HTML `divs` [61].

Furthermore, a large widget may look better if it occupies multiple grid cells on the watch (e.g., the calculation result in Figure 1). Considering that GUI composition is real-time, instead of adding further pressure to the constraint solver, we allow developers to manually specify which widgets should span multiple columns. Such widgets are usually a must in the selection, and a miss often leads to a synthesis error.

3.3 Prototype Implementation

We integrated the algorithms as Jigsaw SDK consists of ~2,000 lines of Python, ~2,000 lines of Kotlin, and ~750 lines of Typescript. The SDK consists of three major components:

The synthesizer (JwSyn) implements the widget selector synthesis algorithm (Section 3.1), in which examples are represented by annotated UIAutomator [26] dumps. Observing that a long chain of nested container widgets slows down the synthesis procedure (which is common in practical dumps), JwSyn compresses the GUI layout by removing container widgets with only one child. Considering that widget attributes except `id` can only play a small role in selection generalization, we implemented JwSyn as using the `id` attribute only by default. However, users can also configure JwSyn to open other attributes.

The server stub (JwStub) is a background service associated with the phone app. JwStub privately maintains an internal activity stack for GUI layout dumps. Delegated events are directly invoked through interfaces like `Activity#dispatchTouchEvent()` with `InputManager` being bypassed (internally maintained GUIs are not sent to the rendering pipeline). The service monitors GUI layout changes, applies the synthesized widget selector, and invokes an Android port of Z3 [75] to solve constraints to compose the selected widgets (Section 3.2). Finally, JwStub is kept alive by starting itself in a detached thread to prevent being killed by the system's task manager.

The watch companion app template (JwCli) can be instantiated with designated app id, name, and icon to yield an installable watch companion app (APK). JwCli contains code for communications with the server stub, rendering composed widgets[6], and pushing on-watch GUI events (e.g., clicks and text inputs) back to the server stub. Developers may also add NativeScript [55] CSS styles to the composed widgets in JwCli for pretty displaying of widgets.

4 EVALUATION

This section presents the quantitative evaluation and results of the widget selection and composition process, followed by a case study of synthesized watch companions for practical usage scenarios. Specifically, we evaluate Jigsaw around the following research questions:

RQ1 (Widget Selection) Is widget selection effective (well generalizing developer annotated widgets) and efficient (completing the synthesis within a reasonable amount of time)?

RQ2 (Widget Composition) Does widget composition appropriately place widgets on the watch screen of limited space?

[6]Widgets with dynamic contents (e.g., `WebView`) cannot be efficiently ported to the resource-limited watch, and are rendered as `ImageViews` in JwCli at present.

RQ3 (Case Study) Can the synthesized companion support simple yet regular user tasks related to the app's core functionalities?

4.1 Experimental Setup

Apps. We evaluate Jigsaw using 16 real-world open-source apps whose users can exploit a smartwatch to perform functional tasks. The first five apps, GoogleCalculator (GC), K9Mail (K9), AntennaPod (AP), Materialistic (MA), and Aard2 (AA), are from frequently occurring subjects in the evaluation of existing Android-app related research [1, 12, 31, 70]. The other 11 are randomly selected popular open-source apps from the Reading, Multimedia, and Internet categories of F-Droid: FDroid (FD), NewPipe (NP), Barinsta (BA), Twidere (TW), Slide (SL), RedReader (RR), NewsBlur (NB), and TrackerControl (TC), NiceFeed (NF), QuickDic (QD), and RadarWeather (RW). We select these 11 apps based on *star*s if they are hosted on Github/Gitlab or *install*s if they otherwise are released on Google Play Store. Specifically, 7/11 selected apps have at ≥1.2K stars and 1/11 have ≥50K installs at the time of writing this paper. We also included TrackerControl, NiceFeed, and RadarWeather because we believe that watch companions will be a great plus for them. We evaluated the latest stable versions[7] of all subjects by the time this paper was written. The supplementary materials contain further information.

Our experiments mainly concern whether Jigsaw can effectively automate the development of smartwatch companions. We found that none of the experimental subjects has a watch counterpart yet, meaning the gap between smartphone app developers and watch apps still exists. We argue that Jigsaw has the potential to bridge this gap for its ease of use: developing a watch companion only requires annotating a few GUI layouts and injecting JwStub to the app with only two lines of code. Thus, the rest of the evaluation focuses on effectiveness and efficiency studies.

RQ1: widget selection. Widget selectors are synthesized on a per-state basis. We identified 39 app states in which functionally critical widgets exist and could be mirrored to the watch as listed in Table 1. We specified a state by a set of state-unique widgets (denoted by its unique ID), i.e., we consider the app at this state if all designated widgets are present. This idea follows DroidBot [45], and developers should have no difficulty in specifying states in this way.

Then we dumped one GUI snapshot for each state and created the GUI snapshot's corresponding hypothetical watch companion (GUI layout on the watch) purely on a developer basis. We annotated the GUI snapshot for positive/negative widget examples to simulate a developer. We provided a *least* number of examples that we feel sufficient for generalization from a human perspective. In total, we provided 155 (3.97 per state on average) positive/negative examples. Annotated GUI layouts and the GUI state information are the only inputs to Jigsaw.

For a quantitative study of the effectiveness of our synthesis algorithm, we further annotated each GUI layout's ground truth widgets to or not to be mirrored on the watch by following the WearOS design principles [25][8]. We annotated 505 positive widgets and 1,234 negative widgets for the 39 states (1,739 ground truth widgets in total). Widgets selected by the synthesized widget selector will be checked against the ground truth for precision, recall, and F1 score.

Our ground truths for the experiments (examples, hypothetical watch companions, and annotated widgets) are all subjective, and the statistics (e.g., precision) cannot directly indicate the absolute effectiveness of Jigsaw. However, we argue that a high precision strongly correlates to the high usefulness of Jigsaw in practice. This is because a high precision indicates that Jigsaw generalizes well from *at least one* developer's perspective.

RQ2: widget composition. Grid synthesis is evaluated as a separate problem. Specifically, for each set of selected widgets (in answering **RQ1**), we created its ground-truth grid by manually positioning each widget in a best-looking way given the GUI layout screenshot and the column number n.

The similarity between a synthesized grid and the ground truth is measured by the distribution of distances between each widget and its ground truth. For a widget w being placed at (r_w, c_w) by Jigsaw and its ground truth (r_w^*, c_w^*), we measure the Euclidean distance

$$\Delta_w = \sqrt{(c_w - c_w^*)^2 + (r_w - r_w^*)^2}$$

and collect the mean and standard deviation

$$\left(\texttt{mean}(\{\Delta_w\}), \texttt{std}^2(\{\Delta_w\})\right)$$

for evaluating the effectiveness of grid synthesis.

Similar to the study for **RQ1**, different developers may provide different ground truths. However, smaller mean and standard deviation do correlate to better grids which look like the best-looking of widgets.

RQ3: case study. To study the real-world applicability of synthesized watch companions, we created 57 user tasks for the 16 apps to simulate real-world usages of a short piece of regular daily task on the watch companion (requires a few GUI events), as listed in Table 2. The tasks are created by reading the app's descriptions on Google Play Store and proposing the core functionalities as if we were the developers.

We simulate a user who interacts with the smartwatch companion (synthesized from the annotated GUI states for answering **RQ1** and **RQ2**) for completing each task. We use the following strict criteria for a task to be successfully performed:

(1) the synthesized widget selector does not miss any functionally important widget,
(2) the widgets are reasonably displayed on the watch,
(3) the state transition of the companion is correct, and
(4) the corresponding goal of the task is accomplished.

All tasks are created independently of the synthesis results. Succeeding cases strongly indicate that the synthesized watch companion is suitable for such a task. Even failed cases (e.g., the erroneously mirrored button in Figure 4) may still be useful to a watch user. Such cases may also be corrected by a developer with moderate effort.

[7] Latest stable version denotes the developer's suggested version on F-Droid by the time of writing this paper.

[8] Widget ID based XPaths in our implementation is expected to generalize well to unseen GUI layouts of the same state because `id` does not change over independent runs. Thus, we did not further annotate GUI layouts to reduce human labor.

Table 1: Effectiveness and Efficiency of Jigsaw. #Ex is the number of examples. #V/#E/#XP are the vertex, edge, and XPath count in the synthesized widget selector. #Exp/#All is the expected and all widget count of our ground truth. T denotes time.

#	App (State)	#Ex.	Selection (Offline)						Composition (Online)		
			#V/#E (#XP)	T (m)	#Exp/#All	Prec	Rec	F1	T (ms)	Mean	Std[2]
1	Aard2 (Bookmarks)	1^+, 1^-	6/25 (1)	<0.01	4/25	1.00	1.00	1.00	8.94	0.00	0.00
2	Aard2 (Dictionaries)	2^+, 1^-	14/66 (2)	<0.01	12/42	1.00	1.00	1.00	18.39	0.50	0.25
3	Aard2 (History)	1^+, 1^-	6/25 (1)	<0.01	8/37	1.00	1.00	1.00	12.18	0.00	0.00
4	Aard2 (Word)	1^+, 0^-	5/18 (1)	<0.01	1/14	1.00	1.00	1.00	7.12	0.00	0.00
5	AntennaPod (Discover)	1^+, 0^-	8/42 (1)	<0.01	12/40	1.00	1.00	1.00	19.26	0.50	0.25
6	AntennaPod (Episode)	4^+, 0^-	35/198 (2)	0.15	7/81	1.00	1.00	1.00	13.86	0.43	0.24
7	AntennaPod (Episodes)	2^+, 1^-	15/75 (2)	<0.01	20/117	1.00	1.00	1.00	36.74	1.00	0.00
8	AntennaPod (Podcast)	6^+, 2^-	31/116 (6)	0.06	16/26	1.00	1.00	1.00	26.87	0.88	0.11
9	AntennaPod (Detail)	3^+, 1^-	23/117 (3)	0.02	11/98	1.00	1.00	1.00	16.23	0.00	0.00
10	Barinsta (Feed)	1^+, 0^-	7/33 (1)	<0.01	15/68	1.00	1.00	1.00	53.11	0.00	0.00
11	Barinsta (People)	7^+, 1^-	49/231 (7)	0.76	12/52	1.00	1.00	1.00	19.78	0.00	0.00
12	Barinsta (Post)	3^+, 1^-	15/54 (3)	<0.01	3/30	1.00	1.00	1.00	15.58	0.00	0.00
13	FDroid (App Detail)	5^+, 0^-	39/204 (4)	0.09	6/19	1.00	1.00	1.00	10.87	2.17	1.81
14	FDroid (Latest)	2^+, 1^-	14/66 (2)	<0.01	30/40	1.00	1.00	1.00	480.97	0.00	0.00
15	GoogleCalculator (Calculator)	3^+, 0^-	17/68 (2)	<0.01	21/38	0.90	1.00	0.95	580.39	1.00	0.00
16	K9Mail (Eamil)	4^+, 2^-	33/185 (4)	0.31	4/18	1.00	1.00	1.00	9.71	0.00	0.00
17	K9Mail (Inbox)	2^+, 0^-	13/58 (2)	<0.01	24/66	1.00	1.00	1.00	49.62	0.00	0.00
18	Materialistic (Main)	3^+, 1^-	18/75 (3)	<0.01	29/91	1.00	1.00	1.00	64.35	3.07	15.44
19	Materialistic (Story)	5^+, 1^-	35/166 (4)	0.02	11/32	0.91	1.00	0.95	17.14	0.70	0.21
20	NewPipe (Trending)	2^+, 1^-	16/84 (2)	<0.01	14/44	1.00	1.00	1.00	24.10	0.00	0.00
21	NewsBlur (Main)	4^+, 1^-	24/100 (4)	<0.01	12/54	1.00	1.00	1.00	22.35	0.17	0.14
22	NewsBlur (Stories)	3^+, 1^-	18/75 (3)	<0.01	21/93	0.86	1.00	0.92	77.62	6.23	3.05
23	NewsBlur (Story)	4^+, 1^-	27/124 (4)	<0.01	4/30	1.00	1.00	1.00	9.43	0.00	0.00
24	NiceFeed (Entry)	3^+, 1^-	20/91 (3)	<0.01	3/11	1.00	1.00	1.00	15.24	0.00	0.00
25	NiceFeed (NewEntries)	2^+, 1^-	14/66 (2)	<0.01	14/27	1.00	1.00	1.00	25.53	0.00	0.00
26	NiceFeed (StarredEntries)	2^+, 1^-	12/50 (2)	<0.01	2/10	1.00	1.00	1.00	9.89	0.71	0.50
27	QuickDic (Dictionary)	3^+, 2^-	18/75 (2)	<0.01	28/37	1.00	1.00	1.00	51.00	1.93	23.28
28	QuickDic (Manager)	5^+, 0^-	35/166 (3)	<0.01	42/56	1.00	0.88	0.93	128.43	6.43	22.77
29	QuickDic (Nav)	3^+, 0^-	14/48 (2)	<0.01	5/5	1.00	1.00	1.00	11.40	0.00	0.00
30	QuickDic (Word)	1^+, 0^-	4/12 (1)	<0.01	1/5	1.00	1.00	1.00	7.98	0.00	0.00
31	RadarWeather (Weather)	4^+, 0^-	26/117 (2)	<0.01	6/32	0.67	1.00	0.80	11.06	1.79	0.79
32	RedReader (/r/art)	3^+, 1^-	35/251 (2)	18.28	6/32	1.00	1.00	1.00	9.75	0.00	0.00
33	RedReader (/r/askreddit)	2^+, 1^-	20/126 (1)	0.44	22/85	1.00	0.92	0.96	33.75	12.41	38.70
34	RedReader (Tags)	2^+, 1^-	15/75 (1)	<0.01	18/42	1.00	1.00	1.00	27.07	10.39	25.24
35	Slide (ALL)	2^+, 0^-	15/75 (2)	<0.01	4/35	1.00	0.80	0.89	9.28	0.00	0.00
36	Slide (Reddit)	3^+, 0^-	22/108 (2)	<0.01	3/46	1.00	1.00	1.00	8.79	0.67	0.22
37	TrackerControl (Main)	3^+, 0^-	22/108 (2)	<0.01	21/36	1.00	1.00	1.00	44.89	0.00	0.00
38	Twidere (Feed)	5^+, 3^-	36/175 (4)	0.04	14/54	1.00	0.93	0.97	24.57	0.00	0.00
39	Twidere (Tweet)	8^+, 6^-	55/257 (5)	0.05	29/71	0.59	1.00	0.74	64.40	3.88	14.22
	Summary	3^+, 1^-	21.3/102.7 (2.6)	0.52	515/1739	0.9724	0.9878	0.9771	53.27	1.4061	3.7750

Summary. Overall, the evaluation consists of 39 widget selector synthesis problems and 57 real-world user tasks for the 16 evaluated apps. For non-deterministic statistical data (e.g., running time) in **RQ1** and **RQ2**, we present the average over three runs. Deterministic values (e.g., synthesis results) are confirmed to be deterministic over all three runs. All experiments were conducted on a OnePlus 6T physical device and a WearOS Round Chin watch emulator (all default settings). The host machine is a quad-core Intel i7-7700 desktop with 32GiB RAM running Ubuntu 20.04.1 LTS, with Android API 28. Detailed descriptions of the experimental subjects (e.g., annotated widgets, ground truth, and synthesized selectors) and results are included in the supplementary materials.

4.2 Results for RQ1: Widget Selection

The "Selection" block of Table 1 (in blue) displays the results of this experiment. For each state, Jigsaw successfully returned a non-trivial widget selector (nonequivalent to `/**/*/`) within 20 minutes (the "T" column). For most (28/39, 71.8%) cases, the synthesis completed in a few seconds. For nearly all (38/39, 97.4%) cases, the synthesis completed within one minute. The most time-consuming case ("/r/art" for RedReader) took 18 minutes to synthesize a considerably complex selector of 35 vertices and 251 edges. The intersected version spaces ($\Omega = \{G\}$) contain over 500K edges.

Nevertheless, we still consider a rare case of 18 minutes as affordable for developers because the synthesis is required only once for an app state.

Applying the synthesized selectors to the GUI layouts yielded 515/1739 selected widgets, where 495 (97.2%) are true positives that are annotated in our ground truth. For most (30/39, 76.9%) GUI layouts, the selected widgets are completely consistent with the ground truth (F1 score = 1.0). For nearly all (36/39, 92.3%) GUI layouts, the selector produced mostly correct results with the ground truth with F1 score > 0.9. The precision exceeds 85% for all cases except "Tweet" of Twidere (17/29, 59%) and "Weather" of RadarWeather (4/6, 67%). For the "Tweet" state, Jigsaw incorrectly selected invisible splitters among other sibling widgets that should instead be selected. Providing a negative example suffices for avoiding this issue. For the "Weather", the app developer failed to add any `id` attribute to the example widget's container widgets[9], and thus the synthesized `/*/*/*/*` overly generally selected redundant sibling widgets.

4.3 Results for RQ2: Widget Composition

The "Composition" block of Table 1 (in green) displays our evaluation results for **RQ2**. Grids are synthesized within a reasonable amount of time. For nearly all (36/39, 92.3%) cases, widget composition is less than 100ms (the "T" column), and for half (21/39, 53.8%) cases, widget composition is even less than 20ms. Selecting more widgets on larger grids generally requires more time. A long composition time (>100ms) happens on the "Dictionary" state of QuickDic (19 widgets, $n = 4$), the "Latest" state of FDroid (28 widgets, $n = 4$), and "Calculator" state of GoogleCalc. (48 widgets, $n = 5$). Such issues can be alleviated by softening the constraints (e.g., by solving the LP relaxation of the integer linear programming) or providing the developer's hint on widget placements. Furthermore, widget composition runs in background (on the smartphone) and will not block user interactions with the watch (but will incur little latency on watch GUI updates).

Concerning the consistency with the ground truth, all selected widgets averagely shift 1.41 (Column Mean) cells from the ground truth position with a variance of 3.78 (Column Std^2). In other words, widgets averagely 1.41 grids away from their "best-looking" positions, but occasionally positioned in a drastically different way. For over half (20/39, 51.3%) states, the shift is zero, i.e., we synthesized exactly the best-looking grid.

For the "worst-looking" grids that have >10 average cell shifts, we further analyzed their causes. We found that the "/r/askreddit" (12.41) and "Tags" (10.39) states of RedReader were induced by UIAutomator which incorrectly set the top and bottom bound boxes of widgets falling on screen boundaries to be 0.

We believe that the results in Sections 4.2 and 4.3 have demonstrated the effectiveness and efficiency of Jigsaw widget selector synthesis and widget composition. Figure 4 displays a few succeeding and failed cases. Regular grids look good on a watch screen (Twidere-1, FDroid). However, sometimes we may overly generalize the selection to including unintended widgets (Twidere-2 and Materialistic). We argue that developers can easily remove them.

[9]This should be an accessibility bug according to the Android documentation [23].

Table 2: Practical Usefulness of Companions. #Rq/#Ev is the number of delegated/all events. Delay displays the average time delay (ms) for delegated events. Log and BW denote the size (KB) of generated logs and the bandwidth (KB/s) when conducting a task, respectively.

#	App (Task)	Succ	#Rq/#Ev	Delay	Log	BW
1	GC (Calc Without Del.)	✓	14/14	123.9	34.1	4.9
2	GC (Calc With Del.)	✓	15/15	115.3	36.5	4.7
3	GC (Clear Results)	✓	15/15	115.3	36.5	7.6
4	FD (Explore Apps)	✓	1/4	99.0	74.1	19.3
5	FD (Check App Info)	✓	6/10	99.8	249.9	32.7
6	FD (Install App)	✓	4/5	141.0	134.9	33.2
7	NP (Browse Trending List)	✓	1/2	147.0	102.0	28.3
8	NP (Open a Trending Video)	✓	2/3	220.0	203.9	33.2
9	TC (Browse Apps)	✓	1/4	92.0	41.6	11.1
10	TC (Toggle Block Tracking)	✓	3/5	99.3	85.0	26.9
11	BA (Browse All Feeds)	✓	1/4	202.0	317.7	64.7
12	BA (Open Feed Post)	✓	3/4	280.0	976.0	193.3
13	BA (Check Account Info)	✓	4/5	188.5	628.4	138.7
14	BA (Check Account Feed)	✓	4/5	252.8	1040.5	211.9
15	TW (Browse Timeline)	✓	1/3	161.0	59.6	12.6
16	TW (Open Tweets)	×	3/4	142.0	322.0	61.7
17	TW (Open Retweets)	✓	4/5	132.0	339.6	62.8
18	TW (Browse Tweets Comm.)	✓	3/4	140.0	165.6	46.6
19	K9 (Explore Inbox)	✓	1/2	32.0	38.3	8.7
20	K9 (Read Email)	✓	3/5	126.3	204.6	37.8
21	SL (Browse ALL Reddits)	✓	1/2	135.0	153.9	31.6
22	SL (Open Reddit)	✓	3/4	116.3	168.3	63.7
23	SL (Goto Next/Prev Reddit)	✓	5/6	129.0	182.7	43.5
24	RR (Browse Tags)	✓	1/2	63.0	3.4	1.2
25	RR (Browse Art Reddits)	✓	3/5	104.3	231.4	76.5
26	RR (Open Reddit)	✓	3/4	196.7	333.6	65.8
27	MA (Explore Stories)	✓	1/2	109.0	21.1	5.7
28	MA (Open Stories)	×	4/5	184.5	829.3	186.0
29	MA (Check Comments)	×	3/8	69.3	34.2	7.2
30	QD (Browse Dictionaries)	×	1/4	85.0	56.9	12.8
31	QD (Download Dictionary)	✓	2/3	161.0	95.1	31.7
32	QD (Open Dictionary)	×	4/6	84.8	13.4	2.4
33	QD (Open Word)	✓	4/5	152.5	34.7	7.9
34	QD (Return to Manager)	✓	6/6	65.5	43.2	7.8
35	AA (See Histories)	×	1/1	25.0	12.2	3.4
36	AA (See Bookmarks)	✓	1/1	80.0	11.7	2.8
37	AA (Browse Dictionaries)	×	1/2	81.0	50.6	14.2
38	AA (Disable Dictionary)	✓	2/4	134.5	101.2	26.5
39	AA (Open Word)	✓	3/3	84.7	43.7	8.3
40	AA (Goto Next/Prev Word)	✓	4/4	206.8	45.6	10.3
41	NB (Browse All Categories)	✓	1/2	98.0	34.8	7.4
42	NB (Browse Category)	✓	3/4	82.0	105.6	25.8
43	NB (Go to Next Category)	✓	4/5	143.8	107.7	18.2
44	NB (Open Story)	×	4/5	113.2	84.7	18.1
45	NF (Browse New Entries)	✓	1/3	131.0	162.3	33.4
46	NF (Goto Starred Entries)	×	4/4	-	-	-
47	NF (Browse Starred Entries)	✓	1/1	95.0	26.4	7.3
48	NF (Open Entry)	✓	3/4	108.3	159.2	47.7
49	RW (Checkout Weather)	✓	1/1	94.0	21.2	4.9
50	RW (Swipe to Next/Prev City)	×	2/2	-	-	-
51	AP (Navigate to Discover)	×	4/4	-	-	-
52	AP (Browse Episodes)	✓	1/3	91.0	69.9	12.3
53	AP (Download Episode)	✓	3/4	215.0	164.4	35.7
54	AP (Explore Discover)	✓	1/4	130.0	121.3	24.8
55	AP (Open Episode)	✓	3/5	105.3	141.4	35.2
56	AP (Open Podcast)	✓	2/6	109.5	66.5	13.9
57	AP (Open Podcast Detail)	✓	1/3	484.0	44.3	11.4
	Summary	46/57	3.18/4.47	132.9	164.2	36.0

4.4 Results for RQ3: Case Study

Table 2 presents the evaluation results. We use the synthesized watch companion (using GUI states in Table 1) to complete the case study. The experiment consists of 57 real-world user tasks with each lasting long for 4 GUI events on average. Among them,

Figure 4: Examples of Synthesized Watch Companions

Jigsaw synthesized watch companions successfully accomplished most (46/57, 80.7%) tasks. Considering that the criteria of success are strict (all functionally important widgets are mirrored, selected widgets are well placed on the watch, and all state transitions are correct), the results are indeed promising.

Since the synthesized watch companion is lightweight (only for communication), the average delay (the round-trip time from pressing a widget on a watch until the GUI is updated[10]) of nearly half (19/46, 41.3%) tasks is <100ms and the bandwidth of some (16/46, 34.8%) tasks are notably <10KB/s. For cases of delay >200ms and bandwidth >50KB/s, we found that the main source is Jigsaw SDK's simple treatment for `ImageView`-like widgets. This treatment simply dumps the pixels of such widgets, causing a large data transmission (especially pixel dumps of `WebViews`) in communication, but this can be alleviated by a more strict resolution adaptation.

We also summarize the failures cases as follows:

(1) *Incorrect state representation* (2/11). We erroneously identified the multi-tab GUI in QuickDic as the same state and displays incorrect set of widgets. Developers should have no difficulty in annotating correct GUI states.
(2) *Missed widgets* (2/11). Our evaluation follows strict criteria that we recognize a task as failed if the widget selector misses any functionally important widget. A miss of such widgets indicates the synthesized companion losing some core functionalities of the app.
(3) *App accessibility bugs* (4/11). The app developers may not tag sufficient accessibility information on their apps. For example, the navigation state of Materialistic cannot be modeled because the `drawer` widget misses an `id` attribute. As suggested by Android's official documentation on accessibility [23], these should be considered accessibility bugs.
(4) *Implementation limitations* (3/11). Finally, Jigsaw has its own implementation limitations. For example, the simple treatment for `WebView` sometimes fails to capture the pixels especially when the content of `WebView` is not fully loaded (Figure 4). In addition, Jigsaw's watch companion template currently does not support events other than click and swipe.

[10]Since the results of an event cannot be predicted, JwStub directly returns once the GUI is detected to update. This sometimes causes the GUIs between the smartphone and the smartwatch to be out of sync when there are animations. However, this can be simply mitigated by a resync.

4.5 Discussions

Usefulness of Jigsaw. Summarizing the results in Tables 1 and 2, our experimental results are encouraging that the Jigsaw's synthesized widget selectors and grids well generalize annotated examples with high precision and recall, and the synthesized watch companion apps are useful in helping users complete real-world user tasks. These results indicate that *automating the creation of smartwatch companions for practical open-source Android apps can be possible.*

Jigsaw may benefit the smartwatch app development community in various ways: First, Jigsaw enables fast prototyping of watch apps. Proof-of-concept watch prototypes can be created with nearly trivial engineering efforts, and the app users can immediately benefit from the convenience of a smartwatch. Second, the workflow of Jigsaw may also enable end-user participated customization of watch companions. Different users may favor a different part of an app's functionalities. We can let users annotate widgets on the GUI layout dumps, and Jigsaw will produce a user-customized watch companion. Finally, we call for future research on smartwatch apps. User-customizable GUI widget mirroring may even be integrated as a system-wide service that continuously migrates *any* app's designated widgets to a smartwatch.

Limitations. First, our implementation is specific to the constraints on the `id` attribute, i.e., assuming that sibling widgets should either be all selected or only specific ones are selected. Therefore, Jigsaw cannot generalize complex constraints like a widget's text description matching a regular expression. Version space algebra can theoretically handle such cases. However, the synthesis could be slow and impractical (and this is why we intentionally preferred a simple algorithm). This paper is mainly for demonstrating the possibility of automatic synthesis of watch companions and accelerating the synthesis can be orthogonal work.

Second, synthesizing styles (e.g., spanable texts, fonts, colors, and borders) is out of the scope of this paper. This is a common practice in the research community of GUI layout generation [6, 8, 11, 38, 41, 49] because style synthesis is a considerably different problem that is more related to the field of computer-human interaction.

Jigsaw is also limited in displaying dynamic contents, e.g., videos and `WebViews`. Currently, JwStub down-samples the dynamic contents (widget)'s screenshot and renders a static image on the watch. Sometimes, JwStub may fail when the dynamic content is not fully loaded (e.g., #28 in Table 2 and Materialistic in Figure 4).

Continuously tracking the changes of dynamic contents is considered too resource-intensive and impractical. Developers may use watch-provided API (e.g., `Overlay`) to display such contents.

Threats to validity. Any human-involved study can incur threats to validity. The first threat to validity is that the evaluated tasks are created by the authors, which may be subject to bias. To best alleviate this threat, we tried our best to read each app's description on F-Droid and Google Play Store (if available), intensively use these apps, and analyze the core functionalities as if we were the developers. We also provide a detailed reproduction step of each task in our supplementary materials. We believe that these efforts are sufficient for future research to replicate the case study and validate the case study results.

A similar threat also exists in the annotation/creation of ground truths. For them, we made our best efforts to read each app's source code and documentation to best ensure that the annotated positive/negative widgets correspond to the core functionalities of the app and the created grids follow the official WearOS app principles [25]. Furthermore, we publicize our data (selected examples and ground-truths) to facilitate future research. Considering that the current precision results show that Jigsaw can indeed produce widgets and layouts correctly matching some developers' expectations, we believe that these results suggest the potential of our algorithm and the possibility of automating the creation of smartwatch companions acceptable and useful.

5 RELATED WORK

Generating GUI layout. GUI layout generation (or synthesis), which aims to automatically generate a GUI layout, is the most relevant domain to ours.

Scrcpy [21] is an open-source tool that generates a GUI layout by faithfully streaming an app's screen from a smartphone to a desktop. Since such streaming is conducted at a pixel level, it does not support smaller screens for example smartwatch. Iuit [69] has considered optimizing a web app's UIs such that the UIs can be automatically adapted to various screen sizes. However, Iuit requires a sufficient number of high-quality smartwatch apps for training a statistical model. This is not practical in the current app store. Sui [3] proposes a framework to adapt UIs among smart-phones, -tablets, and -TVs. However, Sui requires to use the Sui framework when the app is developed and does not support smartwatches. Jigsaw can synthesize watch companions for existing apps that are suitable to use in smartwatches.

The other work attempt to generate the layout from a GUI design. InferUI [8, 41] and Mockdown take developer-provided GUI designs on a single device as input and exploit programming by example to synthesize a cross-device (smartphone) relational GUI layout (e.g., `ConstraintLayout`) which loyally renders all widgets and preserves their relational positions on other devices. Nevertheless, these approaches cannot scale to smartwatches as there lacks an available GUI design. Similar work also considered generating GUI layouts (e.g., `ConstraintLayout`) from GUI designs using deep learning [6, 11, 19, 38, 53, 56, 60]. Compared with black-box approaches, our approach synthesizes an interpretable program which the developer can read and modify.

Furthermore, to the best of our knowledge, our work is the first to synthesize a watch companion for an Android app full automatically and require few examples.

Programming by example. Jigsaw adopts program synthesis to synthesize a widget selector, belonging to the family of programming by examples [32].

Our widget selector is inspired by regular expressions (regex) and regex synthesis [13, 43, 44, 48] in which regexes are synthesized by expanding the regex grammar. Different from such grammar- and enumeration-based approaches which may produce invalid regexes, Jigsaw avoids this by designing and creating the version space of each example.

On the other hand, the algorithm of Jigsaw resembles FlashFill [30] which also exploits a greedy version space algebra to synthesize a regex-like string program. However, the version space of our selector is considerably different from that of FlashFill's and should be carefully designed and intersected.

Other research on Android apps. Different from ours, much prior research on Android concerns the quality of an app. Research like [7, 22, 31, 33, 59, 63] propose to generate test scripts through record and replay. ATM [5] and CraftDroid [46] migrate existing tests to other similar apps; FrUITeR [76] evaluates their migration results. Monkey [24] and followers [20, 29, 50–52, 57, 66, 70] leverage test input generation to trigger more crashes. There is also research concentrating on detecting other non-functional bugs, for example, network [35, 39], data loss [1, 62], and accessibility [12]. Recently, non-crash bugs draw researcher's attention [67, 68]. Besides quality, there is research focusing on security [2, 71, 77], performance [47], and resources [34]. They are generally orthogonal to Jigsaw, and beyond the scope of this paper.

6 CONCLUSION

This paper proposes the push-button synthesis approach to bridging the easily overlooked gap between Android smartphone app developers and smartwatches. Our Jigsaw framework already demonstrated the power of automatically creating watch companions that can help users accomplish real-world tasks.

On the other hand, as a first attempt in this research domain, this paper raises more questions than it solves. To enable a practical watch companion automation for everyone, challenges include but are not limited to: automating the process of example annotation, effective synthesis of complex selectors, relaxing the grid constraints, synthesizing visually satisfactory stylesheets, and synthesizing end-user customizable watch companions. This paper calls for future research along this line.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is supported in part by National Natural Science Foundation of China (Grant #61932021), the Leading-edge Technology Program of Jiangsu Natural Science Foundation (Grant #BK20202001), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn) and Chang Xu (changxu@nju.edu.cn) are the corresponding authors.

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 24th International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA '15)*. Association for Computing Machinery, New York, NY, USA, 83–93. https://doi.org/10.1145/2771783.2771786

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[3] Yuseok Bae, Bongjin Oh, and Jongyoul Park. 2014. Adaptive Transformation for a Scalable User Interface Framework Supporting Multi-screen Services. In *Ubiquitous Information Technologies and Applications*, Young-Sik Jeong, Young-Ho Park, Ching-Hsien (Robert) Hsu, and James J. (Jong Hyuk) Park (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–432.

[4] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 238–249. https://doi.org/10.1145/2970276.2970313

[5] Farnaz Behrang and Alessandro Orso. 2018. Automated Test Migration for Mobile Apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 384–385. https://doi.org/10.1145/3183440.3195019

[6] Tony Beltramelli. 2018. Pix2code: Generating Code from a Graphical User Interface Screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (Paris, France) *(EICS '18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/3220134.3220135

[7] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating Video Recordings of Mobile App Usages into Replayable Scenarios. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 309–321. https://doi.org/10.1145/3377811.3380328

[8] Pavol Bielik, Marc Fischer, and Martin Vechev. 2018. Robust Relational Layout Synthesis from Examples for Android. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 156, 29 pages. https://doi.org/10.1145/3276526

[9] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1987. Occam's razor. *Information processing letters* 24, 6 (1987), 377–380.

[10] Bootstrap. 2021. *Grid System.* https://getbootstrap.com/docs/4.0/layout/grid

[11] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 665–676. https://doi.org/10.1145/3180155.3180240

[12] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 322–334. https://doi.org/10.1145/3377811.3380327

[13] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

[14] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. StoryDroid: Automated Generation of Storyboard for Android Apps. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering* (Montreal, QC, Canada) *(ICSE '19)*. 596–607. https://doi.org/10.1109/ICSE.2019.00070

[15] Xiao Chen, Wanli Chen, Kui Liu, Chunyang Chen, and Li Li. 2021. A Comparative Study of Smartphone and Smartwatch Apps. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1484–1493. https://doi.org/10.1145/3412841.3442023

[16] J. Clement. 2020. App Stores: Number of Apps in Leading App Stores 2020, May 2020.

[17] Ant Design. 2021. *Grid.* https://ant.design/components/grid

[18] Material Design. 2021. *Responsive Layout Grid.* https://material.io/design/layout/responsive-layout-grid.html#columns-gutters-and-margins

[19] Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the 2010 SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1525–1534. https://doi.org/10.1145/1753326.1753554

[20] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-Travel Testing of Android Apps. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 481–492. https://doi.org/10.1145/3377811.3380402

[21] Genymobile. 2021. *scrcpy.* https://github.com/Genymobile/scrcpy

[22] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 72–81.

[23] Google. 2021. *Make Apps More Accessible.* https://developer.android.com/guide/topics/ui/accessibility/apps

[24] Google. 2021. *Monkey.* https://developer.android.com/studio/test/monkey

[25] Google. 2021. *Principles of Wear OS development.* https://developer.android.com/training/wearables/principles

[26] Google. 2021. *UI Automator.* https://developer.android.com/training/testing/ui-automator

[27] Google. 2021. *Watch out for Wear OS at Android Dev Summit 2021.* https://android-developers.googleblog.com/2021/10/wearos-at-ads-21.html

[28] Google. 2021. *Wear OS user interfaces.* https://developer.android.com/training/wearables/user-interfaces

[29] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications via Model Abstraction and Refinement. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 269–280. https://doi.org/10.1109/ICSE.2019.00042

[30] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *SIGPLAN Not.* 46, 1 (Jan. 2011), 317–330. https://doi.org/10.1145/1925844.1926423

[31] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. Sara: Self-Replay Augmented Record and Replay for Android in Industrial Cases. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA '19)*. Association for Computing Machinery, New York, NY, USA, 90–100. https://doi.org/10.1145/3293882.3330557

[32] Daniel Conrad Halbert. 1984. *Programming by example.* Ph.D. Dissertation. University of California, Berkeley.

[33] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet Lightweight Record-and-Replay for Android. *SIGPLAN Not.* 50, 10 (Oct. 2015), 349–366. https://doi.org/10.1145/2858965.2814320

[34] Yigong Hu, Suyi Liu, and Peng Huang. 2019. A Case for Lease-Based, Utilitarian Resource Management on Mobile Devices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3297858.3304057

[35] Peng Huang, Tianyin Xu, Xinxin Jin, and Yuanyuan Zhou. 2016. DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services* (Singapore, Singapore) *(MobiSys '16)*. Association for Computing Machinery, New York, NY, USA, 221–234. https://doi.org/10.1145/2906388.2906419

[36] Allen Hurlburt. 1982. *Grid: A Modular System for the Design and Production of Newpapers, Magazines, and Books.* John Wiley & Sons.

[37] Google I/O. 2014. Designing for Wearables.

[38] Vanita Jain, Piyush Agrawal, Subham Banga, Rishabh Kapoor, and Shashwat Gulyani. 2019. Sketch2Code: Transformation of Sketches to UI in Real-time Using Deep Neural Network. arXiv:1910.08930 [cs.CV]

[39] Xinxin Jin, Peng Huang, Tianyin Xu, and Yuanyuan Zhou. 2016. NChecker: Saving Mobile App Developers from Network Disruptions. In *Proceedings of the 11th European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 22, 16 pages. https://doi.org/10.1145/2901318.2901353

[40] Duling Lai and Julia Rubin. 2019. Goal-Driven Exploration for Android Applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California, USA) *(ASE '19)*. 115–127. https://doi.org/10.1109/ASE.2019.00021

[41] Larissa Laich, Pavol Bielik, and Martin Vechev. 2020. Guiding Program Synthesis by Learning to Generate Examples. In *Proceedings of the 8th International Conference on Learning Representations* (Virtual Conference, Addis Ababa, Ethiopia) *(ICLR '20)*. https://openreview.net/forum?id=BJl07ySKvS

[42] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. 2000. Version Space Algebra and Its Application to Programming by Demonstration. In *Proceedings of the 17th International Conference on Machine Learning* (Stanford, CA, USA) *(ICML '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 527–534.

[43] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the 15th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE '16)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2993236.2993244

[44] Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. 2021. TransRegex: Multi-modal Regular Expression Synthesis by Generate-and-Repair. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering* (Virtual Event, Madrid, Spain) *(ICSE '21)*. 1210–1222. https://doi.org/10.1109/ICSE43902.2021.00111

[45] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a Lightweight UI-Guided Test Input Generator for Android. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering Companion* (Buenos Aires, Argentina) *(ICSE-C '17)*. 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[46] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer across Mobile Apps through Semantic Mapping. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California, USA) *(ASE '19)*. IEEE Press, 42–53. https://doi.org/10.1109/ASE.2019.00015

[47] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE '14)*. Association for Computing Machinery, New York, NY, USA, 1013–1024. https://doi.org/10.1145/2568225.2568229

[48] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing* (Austin, Texas, USA) *(EMNLP '16)*. Association for Computational Linguistics, Austin, Texas, 1918–1923. https://doi.org/10.18653/v1/D16-1197

[49] Dylan Lukes, John Sarracino, Cora Coleman, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. Synthesis of Web Layouts from Examples. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE '21)*. Association for Computing Machinery, New York, NY, USA, 651–663. https://doi.org/10.1145/3468264.3468533

[50] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE '13)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[51] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE '14)*. Association for Computing Machinery, New York, NY, USA, 599–609. https://doi.org/10.1145/2635868.2635896

[52] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA '16)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[53] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2020. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* 46, 2 (2020), 196–221. https://doi.org/10.1109/TSE.2018.2844788

[54] Josef Müller-Brockmann. 1981. *Grid Systems in Graphic Design.* Verlag Gerd Hatje.

[55] NativeScript. 2021. *NativeScript.* https://nativescript.org

[56] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 248–259. https://doi.org/10.1109/ASE.2015.32

[57] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA '20)*. Association for Computing Machinery, New York, NY, USA, 153–164. https://doi.org/10.1145/3395363.3397354

[58] Google Play. 2021. *Google Calculator.* https://play.google.com/store/apps/details?id=com.google.android.calculator

[59] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 571–582. https://doi.org/10.1145/2884781.2884854

[60] Óscar Sánchez Ramón, Jean Vanderdonckt, and Jesús García Molina. 2013. Re-engineering graphical user interfaces from their resource files with UsiResourcer. In *IEEE 7th International Conference on Research Challenges in Information Science* (Paris, France) *(RCIS '13)*. 1–12. https://doi.org/10.1109/RCIS.2013.6577696

[61] HTML Language Reference. 2021. * : Generic Flow Container.* https://www.w3.org/TR/2012/WD-html-markup-20121025/div.html#div

[62] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data Loss Detector: Automatically Revealing Data Loss Bugs in Android Apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA '20)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/3395363.3397379

[63] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse K. Coskun, and Manuel Egele. 2019. RandR: Record and Replay for Android Applications via Targeted Runtime Instrumentation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 128–138. https://doi.org/10.1109/ASE.2019.00022

[64] Statista. 2021. *Smartwatch Market Share of Overall Wearable Market Worldwide in 2018 and 2022, by Operating System.* https://www.statista.com/statistics/910862/worldwide-smartwatch-shipment-market-share

[65] Statista. 2021. *Smartwatch Shipments Forecast Worldwide from 2016 to 2025.* https://www.statista.com/statistics/878144/worldwide-smart-wristwear-shipments-forecast

[66] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-Based GUI Testing of Android Apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE '17)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/3106237.3106298

[67] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully Automated Functional Fuzzing of Android Apps for Detecting Non-Crashing Logic Bugs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 156 (oct 2021), 31 pages. https://doi.org/10.1145/3485533

[68] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and Finding System Setting-Related Defects in Android Apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA '21)*. Association for Computing Machinery, New York, NY, USA, 204–215. https://doi.org/10.1145/3460319.3464806

[69] Hua-Zhe Tan, Wei Zhao, and Hai-Hua Shen. 2018. Adaptive User Interface Optimization for Multi-Screen Based on Machine Learning. In *Proceedings of the 22nd IEEE International Conference on Computer Supported Cooperative Work in Design* (Nanjing, China) *(CSCWD '18)*. 743–748. https://doi.org/10.1109/CSCWD.2018.8465348

[70] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/3377811.3380382

[71] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *Proceedings of the 7th Asia Joint Conference on Information Security* (Tokyo, Japan) *(AsiaJCIS '12)*. 62–69. https://doi.org/10.1109/AsiaJCIS.2012.18

[72] XPath. 2021. *XML Path Language (XPath) 3.1 Specification.* https://www.w3.org/TR/2017/REC-xpath-31-20170321

[73] Yahoo. 2021. *Strategy Analytics: Global Smartwatch Shipments Leap 47 Percent to Pre-Pandemic Growth Levels in Q2 2021.* https://news.yahoo.com/strategy-analytics-global-smartwatch-shipments-103500861.html

[74] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 658–668. https://doi.org/10.1109/ASE.2015.76

[75] Z3. 2021. *Z3 Prover.* https://github.com/Z3Prover/z3

[76] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: A Framework for Evaluating UI Test Reuse. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE '20)*. Association for Computing Machinery, New York, NY, USA, 1190–1201. https://doi.org/10.1145/3368089.3409708

[77] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F. Bissyandé, Jacques Klein, and John Grundy. 2021. On the Impact of Sample Duplication in Machine-Learning-Based Android Malware Detection. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 40 (May 2021), 38 pages. https://doi.org/10.1145/3446905