Robust Relational Layouts Synthesis from Examples for Android

Robust Relational Layout Synthesis from Examples for Android

PAVOL BIELIK, ETH Zürich, Switzerland
MARC FISCHER, ETH Zürich, Switzerland
MARTIN VECHEV, ETH Zürich, Switzerland

We present a novel approach for synthesizing robust relational layouts from examples. Given an application design consisting of a set of views and their location on the screen, we synthesize a relational layout that when rendered, places the components at that same location.

We present an end-to-end system, called InferUI, that addresses the above challenge in the context of Android. The system is based on the following technical contributions: (i) a formalization of the latest and most efficient `ConstraintLayout` class, capturing a rich set of relational constraints, (ii) a set of robustness properties designed to prevent common layout generalization errors, (iii) a synthesis algorithm that produces relational layouts that generalize across multiple screen sizes and resolutions, and (iv) a probabilistic model of constraints that guides the synthesizer towards layouts preferred by developers.

Our evaluation shows that InferUI is practically effective: it successfully synthesizes real world complex layouts obtained from top 500 GitHub and top 500 Google Play Store applications, succeeds in 100% of the cases when synthesizing layouts for a single device, and correctly generalizes 92% of the views across multiple devices, all without requiring additional specifications.

CCS Concepts: • **Software and its engineering** → **Programming by example**; **Automatic programming**; *Interface definition languages*; *Software verification and validation*; • **Human-centered computing** → *Graphical user interfaces*; *User interface design*; • **Computing methodologies** → Machine learning approaches;

Additional Key Words and Phrases: Program synthesis, Programming by example, Relational layouts, Probabilistic models, User interface design, User interface errors

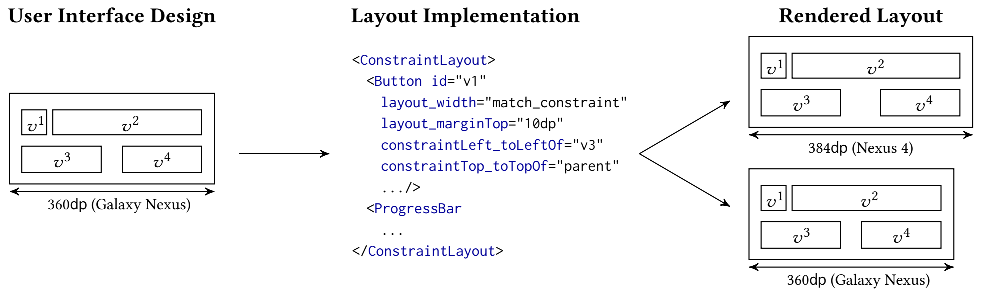
ACM Reference Format:
Pavol Bielik, Marc Fischer, and Martin Vechev. 2018. Robust Relational Layout Synthesis from Examples for Android. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 156 (November 2018), 29 pages. https://doi.org/10.1145/3276526

1 INTRODUCTION

The design and creation of the user interface layouts are core parts of the application development for both desktop and mobile applications. Creating a user interface typically involves a collaboration between a designer, who draws an image of how the user interface should look like, and a developer that implements the design on the desired platform such as Android, iOS, Web or desktop. Concretely, the goal of the developer is to write an implementation of the user interface, referred to as layout, which when rendered on the device places all the views (e.g., buttons, text views, images, etc.) at

Authors' addresses: Pavol Bielik, Department of Computer Science, ETH Zürich, Switzerland, pavol.bielik@inf.ethz.ch; Marc Fischer, Department of Computer Science, ETH Zürich, Switzerland, marcfisc@student.ethz.ch; Martin Vechev, Department of Computer Science, ETH Zürich, Switzerland, martin.vechev@inf.ethz.ch.

2475-1421/2018/11-ART156
https://doi.org/10.1145/3276526



```
<ConstraintLayout>
  <Button id="v1"
    layout_width="match_constraint"
    layout_marginTop="10dp"
    constraintLeft_toLeftOf="v3"
    constraintTop_toTopOf="parent"
    .../>
  <ProgressBar
    ...
</ConstraintLayout>
```

Fig. 1. Main steps of user interface design: (left) the designer draws an image containing four views $v^1, \ldots, v^4$ (e.g., buttons, text views, etc.) how the application user interface should look like, (middle) the developer implements the design for a given platform (e.g., Android), (right) the implementation is rendered on a range of devices with different physical dimensions.

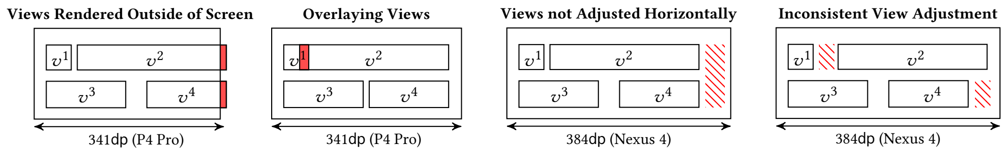


Fig. 2. Illustration of common layout generalization errors, highlighted in red, on devices with different physical dimensions than the device for which the layout was designed.

the same location on the screen as specified by the designer as shown in Fig. 1. For a developer, the task of writing code to generate a given user interface layout is challenging due to the large space of potential designs – many candidate programs can produce visually identical user interface layouts yet some of these programs may fail to generalize well (e.g., when a screen is resized) or have poor performance. Designing robust layouts is a critical factor especially in domains in which layouts are expected to be used across a large number of different contexts. For example, each Android application can be potentially installed on more than 15 000 devices with varying screen resolutions and physical dimensions, all of which need to be considered by the developer during layout implementation.

To illustrate common layout generalization errors, consider the layouts shown in Fig. 2. Given the input design from Fig. 1 the developer can create multiple layouts that all produce the expected results when rendered on Galaxy Nexus device but for various reasons do not generalize well to devices with slightly different screen size such as Nexus 4 or P4 Pro. The leftmost example shows that keeping the absolute view position and size unchanged on a smaller device often results in drawing some of the views outside the screen. Shifting views to the left/right to adjust for the smaller screen size is also not a solution as it can result in overlaying views on top of each other. On a larger device the issue is reversed and not adjusting the views to the screen size leads to visual errors due to resulting blank areas on the screen. The layout that generalizes well should adjust the position of views v^2 and v^4 while also resizing v^2. Note that deciding which views to adjust and how is a hard problem that is context dependent – it depends on other components and their location on the screen. This is where the developer experience is crucial, as it allows to manually create a layout that generalizes to a large number of devices often using only a single example provided by the designer.

To address the gap between visual design and concrete layout implementation, several approaches have been proposed. Given the complexity and time requirements of this task, a common method is to outsource the task to a company that manually creates layouts from images for a high fee [apptype 2018; psd2android 2018; psd2mobi 2018; replia 2018]. Another approach is to try and automate part of the process using better tool support for layout design [Zeidler et al. 2013a,b], generating user interface sketches from real images [Swearngin et al. 2018] or hand drawings [Corrado et al. 2018], sketch based code search for similar layouts [Reiss 2014] or generation of layouts for different screen orientations [Zeidler et al. 2017].

Other approaches try to address the problem by generating layout code directly from images [Beltramelli 2017; Chen et al. 2018; Huang et al. 2016; Nguyen and Csallner 2015]. Their main focus is on the computer vision task required to process raw images and not on the actual layout synthesis. Concretely, (i) they lack a language for expressing layouts [Huang et al. 2016; Nguyen and Csallner 2015] or the language is simple and fails to express many layouts [Beltramelli 2017], (ii) they either do not define a synthesis algorithm or it is implemented as part of a neural network which is non-interpretable, lacks any formal guarantees and can even produce layouts that are syntactically incorrect, and (iii) they return any layout regardless of whether it will generalize to other contexts (i.e., multiple devices) or whether it would be acceptable to a developer.

Our Work. In this work we propose a system, called InferUI, which addresses the above limitations in a principled way. The main idea is to phrase the problem of generating layouts as a program synthesis from examples. Concretely, the examples considered in our work consist of a set of absolute view positions, allowing for a natural way to express the desired design. Then, given a set of views and their absolute positions, InferUI synthesizes a *relational layout* that renders each view according to the absolute view positions. Crucially, we also consider the much harder task of synthesizing a relational layout that generalizes well across *multiple* devices from an input specification consisting of only a *single* device. As there can be a large space of possible solutions, we introduce two additional mechanisms in order to guide the synthesizer towards the desired goal: we present a set of robustness properties that a layout should satisfy (these also prevent common layout generalization errors) and introduce a probabilistic model of constraints learned from existing layouts written by developers (thus, giving preference to more natural layouts).

Main Contributions. Our InferUI system is based on the following technical contributions:

- A formal specification of a set of relational constraints from the latest, most expressive and efficient Android `ConstrainLayout`.
- A new algorithm for synthesizing relational layouts on a *single* device. It succeeds in 100% of cases, bridging the gap between design and layout implementation for one device.
- A new algorithm that synthesizes relational layouts and generalizes well to *multiple* devices. Even when the given specification consist of only a *single* device, the layouts correctly generalize 92% of views in a real world dataset of top 500 Google Play applications.
- A probabilistic model of constraints learned from a large set of layouts written by developers. The model is used to guide the layout synthesis and enables solving complex real world layouts in less than 3 seconds. Further, it allows synthesizing constraints that developers prefer – it correctly predicts the exact constraints written by a developer in 62% of cases.
- A set of robustness properties that capture common visual errors caused by incorrect layouts. We incorporate these properties as part of the synthesis problem, ensuring they are always satisfied by the produced layouts. Further, we use the robustness properties to discover several visual errors in existing Android applications.

2 PRACTICAL BENEFITS OF OUR APPROACH

Our work carries a number of practical benefits when it comes to writing, maintaining and ensuring correctness of relation layouts for Android, including: automation of end-user design, discovering and fixing layout errors as well as porting existing layouts to improve performance. We briefly elaborate on each of these next and discuss at a high level how our approach achieves these goals.

Automating End-User Design. Traditionally, the process of creating any type of application (web, desktop, mobile, etc.) or content (text documents, images) consists of a design phase and an implementation phase. In the design phase the user decides what the result should be whereas the implementation puts the design into effect. In some domains it is possible to lift the implementation to a level that is natural for the user to operate on and hides all internal complexity. For example, consider the task of drawing an image using a stylus pen compared to writing the corresponding vector image directly in SVG format. Even though both approaches represent the result in an SVG format, using the stylus pen is natural, orders of magnitude faster and does not require any technical knowledge about how SVG is implemented.

Unfortunately, even though there are currently almost 3 million Android applications, implementing Android layouts still resembles drawing images by writing SVG format by hand. In particular, Android layouts are represented using XML format where the user needs to know the semantics of several layouts containers (e.g., `RelativeLayout`, `LinearLayout`, `FrameLayout`, etc.), all of their attributes, and how they affect rendering the views on screen. Instead, in our work we hide the implementation complexity of Android layouts from the user and synthesize layouts from a specification that is natural for the user to write – by giving examples of how the views should be positioned on screen.

Avoiding and Finding Layout Errors. A key challenge in layout synthesis and a potential cause of errors is failing to produce layouts that generalize well to a large set of different devices and screen sizes. To address this challenge we developed and formalized a set of robustness properties that good layouts should satisfy. Our synthesis algorithms ensure that all of these properties hold for the generated layouts. Further, as we will demonstrate in the evaluation, these robustness properties are useful beyond synthesis and can be also used to find errors in existing layouts.

Porting Layouts for Better Performance. A major reason why relational layouts were introduced in Android is their rendering performance. Concretely, implementing a given layout using `ConstraintLayout` can result in up to 20% faster rendering speed compared to previous layout implementations (e.g., `LinearLayout` or `RelativeLayout`). However, as `ConstraintLayout` was only recently released, more than 99% of existing layouts in Google Play Store applications are still written using the old layout system. To benefit from this improved performance, developers have no other choice but to manually rewrite their existing layouts. This not only requires considerable amount of time but can introduces errors and visual artefacts. Instead, using our approach developers can automatically synthesize `ConstraintLayout` from their existing layouts. Crucially, the layouts are synthesized not on a "best effort" basic but instead with provable guarantees that the result will visually look the same on all of the supported devices.

To illustrate the reason behind the performance gains, consider the layout shown in Fig. 3 consisting of three views. Here, the designer would like the upper `TextView` to be centered with the `Image` and `TextView` below it. The standard way to implement this on Android is to wrap the bottom two views in a `LinearLayout` which positions them next to each other. Then the `LinearLayout` can be centered with the `TextView` above. This results in deeply-nested layout hierarchies that are slower to render. Instead, the `ConstraintLayout` is more expressive and enables centering a view, in our case the upper `TextView`, in between two other views – the left edge of bottom `Image` and

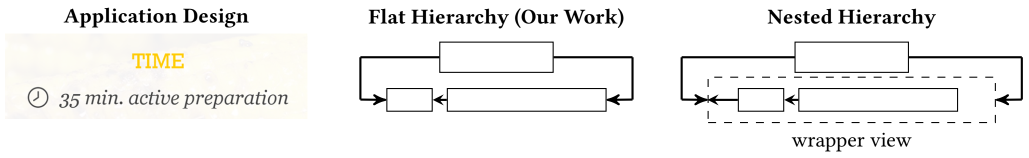


Fig. 3. Same set of views expressed as flat vs nested layout hierarchy. The nested hierarchy uses additional views which purpose is to group other views without being rendered. In contrast, the flat hierarchy uses more expressive layout constraints that removed the need of wrapper views and results in faster layout rendering.

right edge of the bottom `TextView`. This makes the view hierarchy smaller and faster to render by removing the unnecessary `LinearLayout`. Note that the added expressivity also means that `ConstraintLayouts` are harder to synthesize.

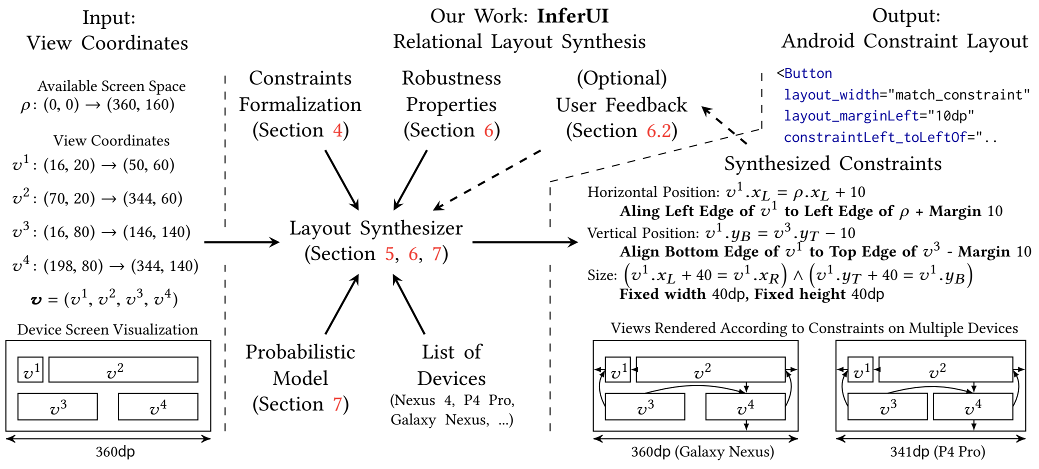
Probabilistic Model of Constraints. Finally, our probabilistic model of constraints is a useful component that can be incorporated in other applications. For example, a common issue with a number of approaches that identify user interface components in images [Beltramelli 2017; Chen et al. 2018; Huang et al. 2016] is that they produce noisy output. A possible approach to reduce this issue would be to incorporate the probabilistic model as an additional term in the loss function, effectively allowing the vision model to adjust the views into positions that produce likely layouts.

3 OVERVIEW

We now present an overview of our approach for synthesizing relational layouts.

Input Specification. In our work we hide the implementation complexity of the relational layouts from the user and synthesize layouts from a natural input specification – the absolute view positions on the screen. The position of each view is specified by two points in the Cartesian coordinate system denoting its top left and bottom right corners respectively. The origin of the coordinate system is located at the top left corner of $(0, 0)$, the positive x-axis points towards the right and the positive y-axis points down. Our approach is independent of the unit of measurement which is generally pixels, or a platform specific equivalent such as density independent pixels (dp) used in Android. As an example, consider the input specification of view $v^1\colon (16, 20) \rightarrow (50, 60)$ which denotes that the top left corner of v^1 is located at position $(16, 20)$ and the bottom right corner at position $(50, 60)$. The coordinates and visualization of sample views v^1, v^2, v^3 and v^4 are shown in Fig. 4 (left). All views are rendered on a canvas we will refer to as content frame ρ, which denotes the allocated portion of the screen where views are rendered.

Relational Layout Synthesis. Given a set of views and their absolute screen position, the goal of relational layout synthesis is to produce a layout that constraints the view size and position such that each view is rendered according to that specification. Consider the constraints generated for view v^1 as shown in Fig. 4 (right). The horizontal constraint specifies that the left edge of view v^1 should be positioned 10 pixels to the right of the left edge of the content frame ρ. This constraint is expressed using a linear equation $v^1.x_L = \rho.x_L + 10$ where $v^1.x_L$ corresponds to left edge of view v^1 and $\rho.x_L$ corresponds to left edge of the content frame. The vertical constraint specifies that the bottom edge of view v^1 should be positioned 10 pixels above the top edge of view v^3. Further, both width and height of the view are 40 pixels. Overall, the output of the synthesis are set of linear constraints that specify the view position relative to other views (denoted as arrows in Fig. 4 bottom right) as well as constraints the specify the view size (i.e., its height and width), both exported as the corresponding source code to be used directly by the developer.


Input:
View Coordinates
Available Screen Space
ρ: (0, 0) → (360, 160)
View Coordinates
v¹: (16, 20) → (50, 60)
v²: (70, 20) → (344, 60)
v³: (16, 80) → (146, 140)
v⁴: (198, 80) → (344, 140)
v = (v¹, v², v³, v⁴)
Device Screen Visualization
360dp
Our Work: InferUI
Relational Layout Synthesis
Constraints Formalization (Section 4)
Robustness Properties (Section 6)
(Optional) User Feedback (Section 6.2)
Layout Synthesizer (Section 5, 6, 7)
Probabilistic Model (Section 7)
List of Devices (Nexus 4, P4 Pro, Galaxy Nexus, ...)
Output:
Android Constraint Layout
Synthesized Constraints
Views Rendered According to Constraints on Multiple Devices
360dp (Galaxy Nexus)
341dp (P4 Pro)

```
<Button
 layout_width="match_constraint"
 layout_marginLeft="10dp"
 constraintLeft_toLeftOf="..
```

Horizontal Position: $v^1.x_L = \rho.x_L + 10$
Aling Left Edge of v^1 to Left Edge of ρ + Margin 10
Vertical Position: $v^1.y_B = v^3.y_T - 10$
Align Bottom Edge of v^1 to Top Edge of v^3 - Margin 10
Size: $(v^1.x_L + 40 = v^1.x_R) \land (v^1.y_T + 40 = v^1.y_B)$
Fixed width 40dp, **Fixed height** 40dp

Fig. 4. Overview of our approach for relational layout synthesis which given a set of views and their absolute positions on the screen synthesizes a layout that renders each view according to that specification. Crucially, the synthesized layout generalizes to a large set of devices on which an application can run in practice.

Once the layout is synthesized it is used to render the views inside of a content frame with different size. The size of the content frame can change for several reasons including using the application on a device with different physical size, using the application in a different mode (i.e., full screen, split screen) or simply reusing the layout in a different application context. As an example, Fig. 4 (bottom right) shows rendering the synthesized layout on a narrower device. In this case, the synthesized layout must correctly render the views on a smaller screen, which can be achieved by moving views v^3 and v^4 closer to each other, decreasing the width of view v^2 and keeping the view v^1 unchanged.

Challenge: Synthesizing Layouts that Generalize across Devices. Creating layout synthesis algorithms that can produce real-world layouts is a hard problem. The first challenge is that the synthesis problem is largely underspecified. Even though our input is a specification for only a single device and application context, in practice, the same layout needs to be rendered potentially on more than 15 000 Android devices with ≈ 100 different density independent screen sizes[1]. Requiring the user to provide and maintain input specifications for all of them is infeasible yet highly desirable – suitable input specification can prevent many generalization errors as illustrated in Fig. 2.

This Work. Our goal is to synthesize a layout based on a *single* specification that generalizes to multiple devices and contexts. To address this challenge we use a combination of several techniques. Firstly, we design a set of robustness properties that capture common generalization mistakes that developers make and consider them during synthesis. Second, we take advantage of existing layouts written by developers and which implicitly capture the constraints that generalize well. For this purpose, we collect a dataset of layouts and learn a probabilistic model of constraints that is used to guide the synthesizer. That is, we produce the most likely layout according to the learned probabilistic model that satisfies the input specification and all our robustness properties. Note

[1]We consider two devices to have the same screen size if their dimensions in pixels are the same (although their physical dimensions might be different). Furthermore, two devices are considered to have the same density independent screen size if their screen sizes contain the same number of density idenpendent pixels used in Android. The statistics are obtained from the Google Developer Console: https://developer.android.com/distribute/index.html

$$View = \langle x_L, x_R, y_T, y_B, y_{baseline} \in \mathbb{Z}^{\geq 0} \rangle$$

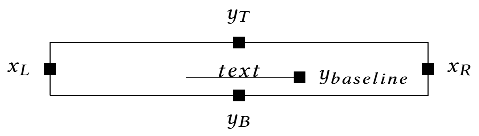


Fig. 5. Definition of a view consisting of five handle points used to relate views to each other (left) and illustration of the handle points on a rendered view (right).

that once the layout is synthesized we directly output the corresponding source code required to implement the layout on Android platform.

As a result, we are able to synthesize layouts that correctly generalize to more than 92% of the views across multiple devices from a single input specification. Finally, if a view does not generalize according to the user expectations we allow the user to specify the desired view position as additional input and re-run the synthesizer.

Challenge: Scaling the Synthesis to Real World Layouts. The second challenge we address is scaling the synthesis algorithm to real world layouts. In particular, given the expressiveness of `ConstraintLayout`, searching directly for a layout that satisfies the input specification and all of our robustness properties does not scale. To address this issue we use the learned probabilistic model of constraints to guide the synthesis search. This allows us to synthesize complex real world layouts in less than 3 seconds.

In what follows we describe our approach to layout synthesis. We start by formalizing the semantics of relational layout constraints in Android (Section 4) together with description of how the layout constraints are solved in order to render a given layout on a device (Section 4.1). Then we introduce our synthesis algorithm for single device (Section 5) which we later generalize to multiple devices (Section 6). Finally, we make both algorithms scalable for complex real world layouts by learning a probabilistic model of constraints that is used to guide the synthesis (Section 7).

4 CAPTURING RELATIONAL CONSTRAINTS

In this section, we define a set of relational constraints that capture the semantics of Android's `ConstraintLayout`.

Views, Handle Points and Relations. We define a *view* as a tuple of five handle points corresponding to left, right, top, bottom edges and text baseline (the vertical position of an imaginary line upon which a line of text rests), illustrated in Fig. 5. Note that the text baseline is defined only for views containing text. For a given view A, we denote these handle points as $A.x_L$, $A.x_R$, $A.y_T$, $A.y_B$ and $A.y_{baseline}$. The handle points allow us to specify relational constraints such as: left edge of view A should be aligned to right edge of view B. That is, we can relate views via their handle points.

Relational Constraints. To relate views via their handle points, three classes of constraints exist – constraints for relative, centering and circular positioning. We define a *constraint* to specify the horizontal position of a view as the following tuple:

$$Constraint = \langle t_h \in \mathcal{C},\ A, B, C \in View, m_L, m_R \in \mathbb{Z}^{\geq 0}, b_h \in \mathbb{R}^{[0,1]}, \alpha \in \mathbb{Z}^{[0,360)}, r \in \mathbb{R} \rangle$$

where t_h is the type of the constraint, A, B and C are views related to each other and m_L, m_R, b_h, α and r are constants corresponding to the left margin, right margin, bias, angle in degrees and distance. Note that constraints typically use only a subset of the constants and views depending

Table 1. Layout constraints that specify the horizontal position (left and right edges) of view A relative to other views (B and/or C). Vertical constraints for top and bottom edges (not shown) are defined analogously.

Relative Positioning Constraints	Dynamic View Size Centering Constraints
$\mathcal{R}_{LL}$: **Align Left of A to Left of B + Margin** $A.x_L = B.x_L + m_L$	$\mathcal{D}_{LL}$: **Center A between Left of B and Left of C + Margin** $A.x_L = B.x_L + m_L \land A.x_R = C.x_L + m_R$
$\mathcal{R}_{LR}$: **Align Left of A to Right of B + Margin** $A.x_L = B.x_R + m_L$	$\mathcal{D}_{LR}$: **Center A between Left of B and Right of C + Margin** $A.x_L = B.x_L + m_L \land A.x_R = C.x_R + m_R$
$\mathcal{R}_{RL}$: **Align Right of A to Left of B + Margin** $A.x_R = B.x_L - m_R$	$\mathcal{D}_{RL}$: **Center A between Right of B and Left of C + Margin** $A.x_L = B.x_R + m_L \land A.x_R = C.x_L + m_R$
$\mathcal{R}_{RR}$: **Align Right of A to Right of B + Margin** $A.x_R = B.x_R - m_R$	$\mathcal{D}_{RR}$: **Center A between Right of B and Right of C + Margin** $A.x_L = B.x_R + m_L \land A.x_R = C.x_R + m_R$

Baseline Constraints	Circular Constraints
$\mathcal{R}_B$: **Align Baseline of A to Baseline of B** $A.y_{baseline} = B.y_{baseline}$	$\mathcal{R}_C$: **Align Center of A to Center of B at an Angle + Distance** $A.x_L + A.x_R = 2r \cdot \sin(\alpha) + (B.x_L + B.x_R)$

Fixed View Size Centering Constraints	
$\mathcal{F}_{LR}$: **Center A between Left of B and Right of C + Margin + Bias**	
$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot (B.x_L + m_L) + b \cdot (C.x_R - m_R) \land$	
$(A = B \land A = \text{ContentFrame}) \Rightarrow (m_L \leq A.x_L - B.x_L \land m_R \leq A.x_R - B.x_R)$	
$\mathcal{F}_{RL}$: **Center A between Right of B and Left of C + Margin + Bias**	
$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot (B.x_R + m_L) + b \cdot (C.x_L - m_R) \land$	
$(A = B \land A = \text{ContentFrame}) \Rightarrow (m_L \leq A.x_L - B.x_L \land m_R \leq A.x_R - B.x_R)$	
$\mathcal{F}_{LL}$: **Center A between Left of B and Left of C + Margin + Bias**	
$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot (B.x_L + m_L) + b \cdot (C.x_L - m_R)$	if $B \neq C$
$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot B.x_L + b \cdot C.x_L \land m_L = 0 \land m_R = 0$	if $B = C$
$\mathcal{F}_{RR}$: **Center A between Right of B and Right of C + Margin + Bias**	
$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot (B.x_R + m_L) + b \cdot (C.x_R - m_R)$	if $B \neq C$
$(1-b) \cdot A.x_L + b \cdot A.x_R = (1-b) \cdot B.x_R + b \cdot C.x_R \land m_L = 0 \land m_R = 0$	if $B = C$

on the constraint type (which also specifies the given constant semantics). The constraints that specify vertical position are defined analogously, except that they use vertical margins and bias (m_T, m_B, b_v) instead of horizontal margin and bias (m_L, m_R, b_h). We now formally define all of the constraints as shown in Table 1, and illustrate them visually in Fig. 6. In total, there are 26 different types of constraints $\mathcal{C}$ that can be applied to a given view in order to specify its location as supported by `ConstraintLayout` version 1.0.2.

Relative Positioning Constraints. A core component in relational layout is constraining the position of a given view relative to another. This can be done either horizontally (e.g., view A is to the right of B) or vertically (e.g., view A is above B). When specifying relative view position we use the handle points corresponding to view edges as defined earlier. For example, the constraint $\mathcal{R}_{LL}$ from Table 1 specifies that the left edge of view A should be aligned to the left edge of view B which results in the constraint $A.x_L = B.x_L + m_L$. Vertical relative constraints are defined analogously to horizontal constraints. The only exception are baseline constraints $\mathcal{R}_B$ which can be used to constraint only the vertical position of a view and do not have a corresponding horizontal version.

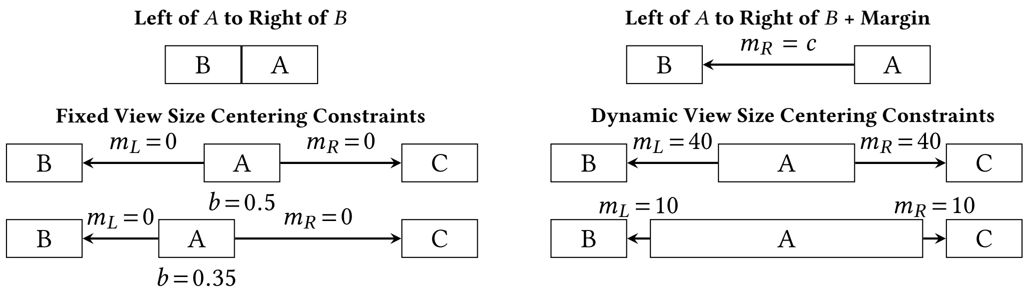


Fig. 6. Illustration of selected relational constraints as formalized in Table 1.

Fixed View Size Centering Constraints. In addition to relating pairs of views using relative position constraints it is also possible to relate view triples. A typical example is horizontally centering a view on the screen as illustrated in Fig. 6. Here, we would like to express that view A should be horizontally centered in between views B and C instead of relating it only to the left (or right) view using a margin. For this purpose, we define centering constraints as shown in the Table 1. Without *margins*, the default *bias*, $b = 0.5$, view A is always centered in between the corresponding handle points of views B and C (note that B and C can refer to the same view). The *margins* have the same semantics as in relative positioning constraints, only that now there are both left and right margins (or top and bottom).

The *bias* attribute $b \in \mathbb{R}^{[0,1]}$ controls the preference for positioning in between views B and C. The intuition is that bias specifies where to position the view on the line segment between both handle points. For instance, for $b = 0.35$, the view will be positioned at 35% of the line segment length as illustrated in Fig. 6. If the bias is set to the minimum (i.e., $b = 0$) the view A is positioned directly to the right of B. Note that even if bias is set to minimum (or maximum) the margins still apply. That is, the centering constraint with $b = 0$ and $m_L = 10$ will position view A at distance 10 from the right edge of view B.

Finally, in order to accurately model the semantics of Android's layout system implementation we strengthened the constraints. In particular, for constraints $\mathcal{F}_{LL}$ and $\mathcal{F}_{RR}$ the margins are ignored if view A is centered with a single handle point (i.e., if $B = C$). Further, for constraints $\mathcal{F}_{LR}$ and $\mathcal{F}_{RL}$ we discovered a bug in the Android layout solver that results in rendering view A incorrectly (its position is shifted by one pixel). This happens in case view A is related to the content frame (a view representing available device screen size) and the margins are larger than its distance from the content frame.

Circular Constraints. Circular position constraints allow relating the center of two views at an angle α and a distance r. Using circular constraints we can easily express that two views are related at an angle $\alpha = 45°$ without having to compute the corresponding margins manually.

Dynamic View Size Centering Constraints. So far all the constraints assumed that the width and height of the view are known. This is because in order to render the view on the screen we need to compute the position of all of its edges and not only one of them or its center. To support dynamic view sizes the constraint has to relate both horizontal (or vertical) handle points. For example, the effect of constraint $\mathcal{D}_{RL} : A.x_L = B.x_R + m_L \wedge A.x_R = C.x_L + m_R$ with $m_L = 10, m_R = 10$ is that view A is rendered between right and left edges of views B and C respectively while spanning the whole space in between them (except for margins) as illustrated in Fig. 6.

$$\psi_{layout}(\rho \in View, \boldsymbol{c} \subset Constraint, \boldsymbol{s} \subset Size) = \phi_{position} \wedge \phi_{size} \wedge \phi_{constraints}$$

$$\phi_{position} \overset{\text{def}}{=} \left(x_L^0 = \rho.x_L\right) \wedge \left(x_R^0 = \rho.x_R\right)$$

$$\phi_{constraints} \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{c}|} [\![\boldsymbol{c}^i]\!] \qquad \phi_{size} \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{s}|} \begin{cases} x_L^i + \boldsymbol{s}^i.width = x_R^i & \text{if } \boldsymbol{s}^i.t_h^i = \texttt{Fixed} \\ x_R^i - x_L^i \geq 0 & \text{otherwise} \end{cases}$$

Fig. 7. Function ψ_{layout} that specifies absolute view positions from relational layout constraints $\boldsymbol{c}$ and view sizes $\boldsymbol{s}$ (where $|\boldsymbol{c}| = |\boldsymbol{s}|$) by encoding this problem as a set of linear equations.

View Size. The view size needs to be specified in addition to relative, circular and fixed size centering constraints in order to compute the absolute positions of all view handle points. The view size is defined as follows:

$$Size = \langle t_h, t_v \in \{\texttt{Fixed}, \texttt{MatchConstraint}\}, width, height \in \mathbb{Z}^{\geq 0} \rangle$$

that is, view size is either fixed (denoted as `Fixed`) in which case the view size is a constant or dynamically computed as specified by the constraints (denoted as `MatchConstraint`). Note that it is allowed for a view to have one dimension fixed while the other dimension is computed dynamically.

4.1 Layout Constraint Solving

Having formalized the relational constraints and view sizes we now describe how they are used to compute absolute view positions. This will be a necessary component for synthesizing layouts that generalize to multiple devices presented in Section 6. Given a set of views with associated constraints and view sizes, we compute the absolute view positions in two steps: (i) we encode view constraints and view sizes as a set of linear equations (formalized in Table 1) where free variables correspond to view handle points, and (ii) we find the satisfying assignment to the free variables by solving the resulting linear equations. In our case, this assignment captures the absolute positions of each view's handle points corresponding to the left, right, top and bottom edges. Note that because all of our constraints are relational (i.e., they only specify the position of a view relative to other views), one may obtain many different satisfying assignments some of which will not match the desired absolute view positions. This is undesirable as it results in non-deterministic view position. Because of this, a so called content frame (typically spanning the available screen size) is included as an additional view with fixed absolute coordinates. Then, if all views are transitively related to the content frame and the relations do not contain cycles, the resulting system of linear equations has exactly one satisfying assignment. Throughout the paper, when indexing views we use ρ or index zero (e.g., v^0) to refer to the content frame and indices 1 and above when referring to other views. Further, for all of our algorithms we simplify the presentation and remove clutter by encoding only the horizontal constraints (the vertical constraints are defined analogously).

Encoding Relational Constraints. Fig. 7 defines how to encode relational constraints as a conjunction of linear equations. We have three kinds of equations: $\phi_{position}$, ϕ_{size} and $\phi_{constraints}$. Here, $\phi_{position}$ specifies the absolute position of the content frame. As this position is known, we simply assign the concrete values to the content view handle points. Equation ϕ_{size} restricts the view size. If the size is fixed it is enforced using $x_L^i + \boldsymbol{s}^i.width = x_R^i$ which specifies that the distance between left and right handle points is equal to the width of the view. If the size is computed dynamically we only enforce that it is non negative. Finally, $\phi_{constraints}$ encodes the actual constraints over the view handle points. We use $[\![\boldsymbol{c}^i]\!]$ to denote evaluation of constraint $\boldsymbol{c}^i$ which returns the logical formula associated with $\boldsymbol{c}^i$ according to its definition from Table 1.

Input: Constraints & Size

$$\boldsymbol{c}_h^1 = \langle t_h = \mathcal{R}_{LL}, A = v^1, B = v^0, m_L = 10\rangle$$
$$\boldsymbol{s}_h^1 = \langle t_h = \texttt{Fixed}, width = 260\rangle$$

$$\boldsymbol{c}_h^2 = \langle t_h = \mathcal{D}_{LR}, A = v^2, B = v^1, C = v^0, m_L = m_R = 10\rangle$$
$$\boldsymbol{s}_h^2 = \langle t_h = \texttt{MatchConstraint}\rangle$$

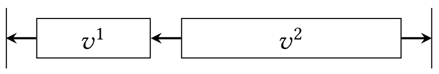
Constraint System

$$\psi_{layout}((0,720),(\boldsymbol{c}_h^1,\boldsymbol{c}_h^2),(\boldsymbol{s}_h^1,\boldsymbol{s}_h^2))$$
$$\phi_{constraints} \stackrel{\text{def}}{=} x_L^1 = x_L^0 + 10 \land x_L^2 = x_R^1 + 10 \land x_R^2 = x_R^0 - 10$$
$$\phi_{position} \stackrel{\text{def}}{=} x_L^0 = 0 \land x_R^0 = 720$$
$$\phi_{size} \stackrel{\text{def}}{=} x_L^1 + 260 = x_R^1 \land x_R^2 - x_L^2 \geq 0$$

Solution

A satifying assigment to x_L^1, x_R^1, x_L^2, x_R^2 in ψ_{layout}

$x_L^1 = 10, x_R^1 = 270$



$x_L^2 = 280, x_R^2 = 710$

Fig. 8. An example illustrating how the relational constraints are encoded as a set of linear equations as specified by ψ_{layout}. The solution of the formula specifies the absolute position of views on the screen.

Example. Fig. 8 illustrates the encoding of the relational layout constraints according to Fig. 7 on a simple example consisting of two views. The constraint $\boldsymbol{c}_h^1$ specifies that left edge of the first view is to the right of left edge of the content frame with margin 10. The constraint $\boldsymbol{c}_h^2$ specifies that the second view should be positioned between the right edges of the first view and the content frame. The width of the first view is fixed while the width of the second view is computed dynamically allowing it to span all of the remaining available space on the screen. By solving the resulting formula we can compute the absolute positions of all views as shown at the bottom of Fig. 8.

5 SINGLE DEVICE RELATIONAL LAYOUT SYNTHESIS

So far we defined how to compute absolute view positions given relational constraints (Section 4.1). We now discuss the inverse problem of generating these constraints given the absolute view positions and sizes (as provided by a user). In particular, for each view we are interested in generating one constraint that controls its horizontal position and one constraint for its vertical position. We need at least one constraint (for either axis) as otherwise the view position is unconstrained and cannot be computed. Moreover, exactly one constraint is also sufficient because if multiple constraints are specified then they are either redundant or unsatisfiable.

Problem Statement. The layout synthesis problem is defined as follows:

Input: A set $\boldsymbol{v} \subset View$ of N views with specified absolute positions on the screen defining where the views should be rendered. A content frame $\rho \in View$ defining the screen size.

Output: A set of N view sizes $\boldsymbol{s} \subset Size$ and N horizontal and vertical constraints $\boldsymbol{c}_h, \boldsymbol{c}_v \subset Constraint$ (one for each input view) where $\boldsymbol{v} \models \psi_{layout}(\rho, \boldsymbol{c}_h, \boldsymbol{c}_v, \boldsymbol{s})$.

That is, we need to define a synthesizer that for a given screen size ρ and absolute positions of all views finds constraints and view sizes whose solution (as specified by ψ_{layout}) matches the input. Such a synthesizer would then automatically compute the constraints and view sizes that a developer has to currently write by hand. In the next section we will show how to extend the synthesizer so that it generalizes across multiple screen sizes (Section 6 and Section 7) and develop a technique to make the synthesizer scale to a large number of views (Section 7).

$$\psi_{single\_syn}(\rho \in View, \boldsymbol{v} \subset View) = \phi_{position} \land \phi_{valid} \land \phi_{constraints} \land \phi_{acyclic}$$

$$\phi_{position} \overset{\text{def}}{=} \left(x_L^0 = \rho.x_L\right) \land \left(x_R^0 = \rho.x_R\right) \land \left(\bigwedge_{i=1}^{|\boldsymbol{v}|} x_L^i = v^i.x_L \land x_R^i = v^i.x_R \right)$$

$$\phi_{valid} \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} m_L^i \geq 0 \land m_R^i \geq 0 \land 0 \leq b_h^i \leq 1 \land 0 \leq \alpha^i < 360 \land r^i \geq 0$$

$$\phi_{constraints} \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} \left(\bigwedge_{k=0}^{|\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|} \boldsymbol{g}_k^i \Rightarrow [\![\boldsymbol{c}_k^i]\!] \right) \land \boldsymbol{g}_0^i + \cdots + \boldsymbol{g}_{|\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|}^i = 1$$

$$\phi_{acyclic} \overset{\text{def}}{=} (\rho.d = 0) \land \left(\bigwedge_{i=1}^{|\boldsymbol{v}|} \boldsymbol{v}^i.d > 0 \land \bigwedge_{k=0}^{|\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|} \boldsymbol{g}_k^i \Rightarrow \boldsymbol{v}^i.d = \begin{cases} [\![\boldsymbol{c}_k^i.B]\!].d + 1 & \text{if } t_h^i \in \mathcal{R} \\ [\![\boldsymbol{c}_k^i.B]\!].d + [\![\boldsymbol{c}_k^i.C]\!].d + 1 & \text{otherwise} \end{cases} \right)$$

Fig. 9. Synthesis algorithm that given a set of absolute view positions $\boldsymbol{v}$ and a content frame ρ computes suitable constraints and view sizes that render the views at the same absolute positions as specified by $\boldsymbol{v}$.

Relational Layout Synthesis. Our synthesis algorithm is shown in Fig. 9. We encode the problem as a logical formula ψ_{single_syn} ranging over boolean, integer and real valued variables. A model of ψ_{single_syn} determines which constraints and view sizes should be applied such that we solve the problem statement above. We divide the formula into four parts $\phi_{position}$, ϕ_{valid}, $\phi_{constraints}$ and $\phi_{acyclic}$ that are described next.

Here, $\phi_{positions}$ encodes the input specification by setting the handle points based on the input views $\boldsymbol{v}$ and content frame ρ. Then, ϕ_{valid} defines domain of constants which are allowed to be used by the (to be synthesized) relational constraints. In particular, margins m_L, m_R and distance r have to be non-negative, bias b_h has to be between zero and one, and the angle is a valid degree. The formula $\phi_{acyclic}$ encodes that constraint relations are acyclic. Acyclic relations are required as otherwise the constraints do not specify a unique solution when solving for absolute view positions (since views depend transitively on themselves). To encode acyclic relations we assign an integer variable $\boldsymbol{v}^i.d > 0$ to each view $\boldsymbol{v}^i$. The intuition behind each $\boldsymbol{v}^i.d$ is that it captures distance from the content frame, where the content frame has a distance of zero (i.e., $\rho.d = 0$). Then, the distance of a view is defined as one plus the sum of distances of the views it relates to (where $[\![\boldsymbol{c}_k^i.B]\!]$ is used to denote view B associated with constraint $\boldsymbol{c}_k^i$). Such encoding efficiently disallows cycles since introducing a cycle will result in an unsatisfiable assignment to distance variables.

The $\phi_{constraints}$ encodes the constraints for each view. Let $\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho)$ denote a set of all admissible constraints for a given view $\boldsymbol{v}^i$. This set is obtained by instantiating each constraint type (defined in Table 1) with view $\boldsymbol{v}^i$ as source (i.e., $A = \boldsymbol{v}^i$) and all the other views including content frame as possible targets (i.e., $B, C \in \{\rho \cup \boldsymbol{v} \setminus \{v^i\}\}$). We then associate a boolean variable $\boldsymbol{g}_k^i$ with each admissible constraint $\boldsymbol{c}_k^i$ for that view and add the implication $\boldsymbol{g}_k^i \Rightarrow [\![\boldsymbol{c}_k^i]\!]$. The interpretation of $\boldsymbol{g}_k^i$ is such that if it is true, the constraint $\boldsymbol{c}_k^i$ is activated, and will be returned as part of the solution. Finally, we enforce that for each view exactly one horizontal constraint is synthesized by restricting the sum of all $\boldsymbol{g}_k^i$ for a given view $\boldsymbol{v}^i$ to be equal to 1[2].

[2]In our implementation we encode this constraint using $PbEq$ function, which is an optimized implementation of pseudo-boolean relations of type $k_1 * p_1 + \cdots + k_n * p_n = k$ provided by Z3 Solver. In our case $k_1, \ldots, k_n = 1$ and $k = 1$.

$$\psi_{multi\_syn}(\rho \in View, \boldsymbol{d} \subset View, \boldsymbol{v} \subset View) = \psi_{single\_syn}(\rho, \boldsymbol{v}) \rightarrow \langle \boldsymbol{c}, \boldsymbol{s} \rangle \wedge \bigwedge_{k=1}^{|\boldsymbol{d}|} \psi_{gen}(\boldsymbol{d}_k, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{s})$$

$$\psi_{gen}(d, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{s}) \overset{\text{def}}{=} \boldsymbol{v}_d \models \psi_{layout\_syn}(d, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{s}) \wedge$$
$$\phi_{inside\_screen}(d, \boldsymbol{v}_d) \wedge \phi_{pixel\_perfect}(\boldsymbol{v}_d) \wedge \phi_{preserve\_aspect\_ratio}(\boldsymbol{v}, \boldsymbol{v}_d) \wedge$$
$$\phi_{preserve\_order}(\boldsymbol{v}, \boldsymbol{v}_d) \wedge \phi_{preserve\_centering}(\boldsymbol{v}, \boldsymbol{v}_d) \wedge \phi_{preserve\_margins}(\boldsymbol{v}, \boldsymbol{v}_d)$$

$$\psi_{layout\_syn}(d \in View, \boldsymbol{v} \subset View, \boldsymbol{c} \subset Constraint, \boldsymbol{s} \subset Size) = \phi_{position} \wedge \phi_{size} \wedge \phi_{constraints}$$

$$\phi_{position} \overset{\text{def}}{=} \left(\boldsymbol{v}_d^0.x_L = d.x_L\right) \wedge \left(\boldsymbol{v}_d^0.x_R = d.x_R\right)$$

$$\phi_{size} \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} \left(\bigwedge_{k=0}^{|C(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|} \boldsymbol{g}_k^i \Rightarrow \begin{cases} \boldsymbol{v}^i.x_R - \boldsymbol{v}^i.x_L = \boldsymbol{v}_d^i.x_R - \boldsymbol{v}_d^i.x_L & \text{if } \boldsymbol{c}^i.t_h^i \notin \mathcal{D} \\ true & \text{otherwise} \end{cases} \right)$$

$$\phi_{constraints} \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} \left(\bigwedge_{k=0}^{|C(\boldsymbol{v}^i, \boldsymbol{v}, \rho)|} \boldsymbol{g}_k^i \Rightarrow [\![\boldsymbol{c}_k^i]\!]^d \right)$$

Fig. 10. Synthesis algorithm that ensures that the synthesized constraints generalize to a set of devices $\boldsymbol{d}$.

View Size. Note that the synthesis algorithm ψ_{single_syn} does not depend on the view size. Instead, the view size can be determined *after* a satisfying assignment to the synthesis formula is found. Concretely, if the synthesized constraint is of type $\mathcal{D}_{LL}, \mathcal{D}_{LR}, \mathcal{D}_{RL}$ or $\mathcal{D}_{RR}$ then the view size is of type `MatchConstraint` and of type `Fixed` otherwise. This is possible because during synthesis, the view size is known as the input $\boldsymbol{v}$ already specifies all handle points.

6 ROBUST LAYOUT SYNTHESIS: GENERALIZING LAYOUTS TO MULTIPLE DEVICES

In this section we build upon the synthesis algorithm ψ_{single_syn} from Section 5 and show how to extend it to synthesize constraints that generalize across multiple devices. This functionality is extremely useful as in practice developers have to consider a large number of devices with different screen sizes and resolutions on which an application might be rendered. For example, there are more than 15 000 device models with almost 100 different screen sizes (even after an adjustment using density independent pixels) that one needs to consider when developing an Android application.

Compared to ψ_{single_syn} which takes only a single device ρ as input, the algorithm to support multiple devices takes a list of devices $\boldsymbol{d} \subset View$ to be considered (or alternatively, a maximum allowed resize ratio of the device ρ). However, note that the input specification still consists of absolute view positions $\boldsymbol{v}$ only for a single device ρ and not for all the devices $\boldsymbol{d}$. As a result, the synthesis problem is severely underspecified as we do not have any specification for the additional devices. We address the under specification issue using two techniques: (i) by designing a set of properties that "good layouts" should satisfy (described in this section), and (ii) by learning from a large set of layouts already written by developers (described in Section 7).

Our synthesis algorithm ψ_{multi_syn} supports multiple devices and is shown in Fig. 10. It consists of three parts. First, using ψ_{single_syn}, we compute formulas for $\boldsymbol{c}$ and $\boldsymbol{s}$ that satisfy the input specification $\boldsymbol{v}$ on device ρ. Second, using $\boldsymbol{c}$ and $\boldsymbol{s}$ from the first step we produce a view layout $\boldsymbol{v}_d$ on each device d, that is, $\boldsymbol{v}_d \models \psi_{layout\_syn}(d, \boldsymbol{c}, \boldsymbol{s})$. Finally, for each device we check that the views $\boldsymbol{v}_d$ satisfy a set of robustness properties. All three steps are in fact encoded as a single logical formula, the solution of which specifies the desired constraints and view sizes.

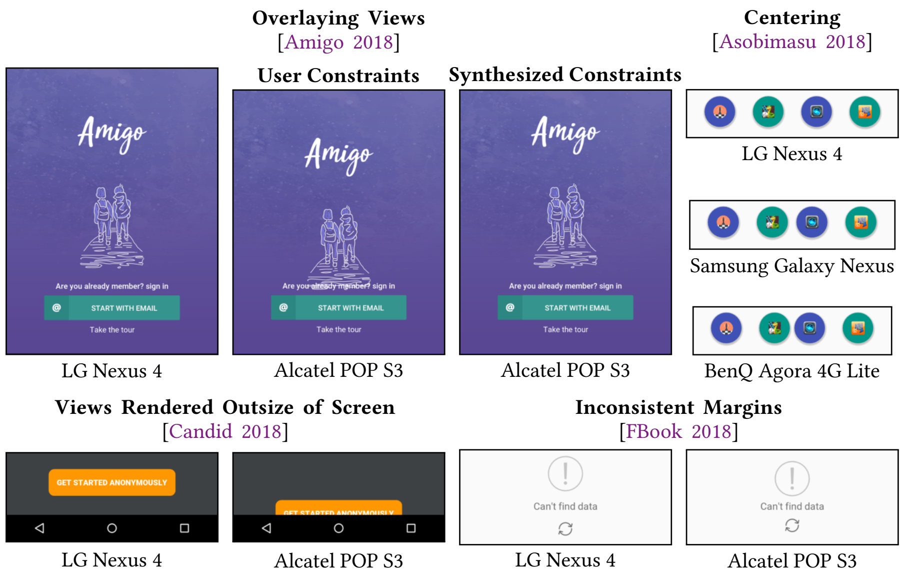


Fig. 11. Illustration of visual errors that arise from using applications on different physical devices that designed for. In these examples, the reference device with the correct design is LG Nexus 4.

Before formalizing the robustness properties we describe the encoding of ψ_{layout_syn}. The idea behind ψ_{layout_syn} is similar to ψ_{layout} defined in Fig. 7 except that now our goal is to encode both layout and synthesis within a single logical formula. For this purpose ψ_{layout_syn} introduces a fresh set of free variables $\boldsymbol{v}_d$ denoting views rendered on a device d. Then, it reuses the boolean variables $\boldsymbol{g}_k^i$ defined in ψ_{single_syn} to restrict the size and apply constraints over $\boldsymbol{v}_d$ (here $[\![\boldsymbol{c}_k^i]\!]^d$ denotes evaluating the constraint $\boldsymbol{c}_k^i$ over a set of views $\boldsymbol{v}_d$).

6.1 Robustness Properties

We designed a set of general properties that good layouts should satisfy and that prevent common errors made by developers. Encoding these properties as part of the synthesis formula allows us to synthesize layouts that generalize well to multiple devices even though their input specification is not available. As an example, consider Fig. 11 which shows four existing applications rendered on a device they were designed for (LG Nexus 4) as well as on other devices. Using the applications on a device with smaller height than designed leads to overlaying two views containing text and image for the Amigo application, rendering a button partially out of the screen in the Candid application as well as visual artefacts caused by stretching common margins used between views in the FBook application. Similarly, using the application on a device with smaller/larger width than designed can lead to errors as the one found in the Asobimasu application. In the remainder of the section we describe and formalize the generalization properties required to avoid common layout errors (including those shown in Fig. 11) and how we encode those in ψ_{multi_syn}.

Preserve Order. The $\phi_{preserve\_order}$ property ensures that when views $\boldsymbol{v}$ are rendered on a different device, their relative order stays the same. That is, if a view A is to the left of view B on a device ρ, we expect the same to hold for all devices. Concretely, we add constraints for each pair of view handle points that ensure their ordering is preserved. This property enforces that a layout producing the visual error in Amigo is not allowed. Instead, the synthesized constraints will prevent overlaying the views and move the application logo and name upwards instead of downwards on a smaller screen (shown in Fig. 11).

$$\phi_{preserve\_order}(\boldsymbol{v}, \boldsymbol{v}_d) \overset{\text{def}}{=} \bigwedge \left\{ \begin{array}{l} aligned_{LL}(\boldsymbol{v}^i,\boldsymbol{v}^j,\boldsymbol{v}_d^i,\boldsymbol{v}_d^j) \wedge aligned_{LR}(\boldsymbol{v}^i,\boldsymbol{v}^j,\boldsymbol{v}_d^i,\boldsymbol{v}_d^j) \\ aligned_{RL}(\boldsymbol{v}^i,\boldsymbol{v}^j,\boldsymbol{v}_d^i,\boldsymbol{v}_d^j) \wedge aligned_{RR}(\boldsymbol{v}^i,\boldsymbol{v}^j,\boldsymbol{v}_d^i,\boldsymbol{v}_d^j) \end{array} \middle| \begin{array}{l} \forall i, j. \\ 0 \le j < i \le N \end{array} \right\}$$

$$aligned_{LR}(\boldsymbol{v}^i, \boldsymbol{v}^j, \boldsymbol{v}_d^i, \boldsymbol{v}_d^j) \overset{\text{def}}{=} \begin{cases} \boldsymbol{v}_d^i.x_L = \boldsymbol{v}_d^i.x_R & \text{if } \boldsymbol{v}^i.x_L = \boldsymbol{v}^i.x_R \\ \boldsymbol{v}_d^i.x_L < \boldsymbol{v}_d^i.x_R & \text{if } \boldsymbol{v}^i.x_L < \boldsymbol{v}^i.x_R \\ \boldsymbol{v}_d^i.x_L > \boldsymbol{v}_d^i.x_R & \text{if } \boldsymbol{v}^i.x_L > \boldsymbol{v}^i.x_R \end{cases}$$

Preserve Margins. A standard design technique is to layout two views within a certain margin of each other. For example, views are aligned to the screen border typically with a margin of 16 pixels and the spacing between two views is typically 8 pixels (or a multiple of 8). The $\phi_{preserve\_margins}$ property ensures that such margins are preserved across multiple devices. In our implementation the margins we preserve, denoted $\mathcal{M}$, correspond to the most commonly used values by developers.

$$\phi_{preserve\_margins}(\boldsymbol{v}, \boldsymbol{v}_d) \overset{\text{def}}{=} \bigwedge \left\{ \begin{array}{ll} \boldsymbol{v}^i.x_L - \boldsymbol{v}^j.x_L = \boldsymbol{v}_d^i.x_L - \boldsymbol{v}_d^j.x_L & \text{if } \left|\boldsymbol{v}^i.x_L - \boldsymbol{v}^j.x_L\right| \in \mathcal{M} \\ \boldsymbol{v}^i.x_R - \boldsymbol{v}^j.x_L = \boldsymbol{v}_d^i.x_R - \boldsymbol{v}_d^j.x_L & \text{if } \left|\boldsymbol{v}^i.x_R - \boldsymbol{v}^j.x_L\right| \in \mathcal{M} \\ \boldsymbol{v}^i.x_L - \boldsymbol{v}^j.x_R = \boldsymbol{v}_d^i.x_L - \boldsymbol{v}_d^j.x_R & \text{if } \left|\boldsymbol{v}^i.x_L - \boldsymbol{v}^j.x_R\right| \in \mathcal{M} \\ \boldsymbol{v}^i.x_R - \boldsymbol{v}^j.x_R = \boldsymbol{v}_d^i.x_R - \boldsymbol{v}_d^j.x_R & \text{if } \left|\boldsymbol{v}^i.x_R - \boldsymbol{v}^j.x_R\right| \in \mathcal{M} \end{array} \middle| \begin{array}{l} \forall i, j. \\ j \ne i \\ 1 \le i \le N \\ 0 \le j \le N \end{array} \right\}$$

Preserve Centering. The $\phi_{preserve\_centering}$ property ensures that when views $\boldsymbol{v}$ are centered on device ρ, they will also be centered on all the other devices in $\boldsymbol{d}$. Note that for this property we need to consider all triples of views that can be centered.

$$\phi_{preserve\_centering}(\boldsymbol{v}, \boldsymbol{v}_d) \overset{\text{def}}{=} \bigwedge \left\{ \begin{array}{ll} centered_{LL}(\boldsymbol{v}_d^i,\boldsymbol{v}_d^j,\boldsymbol{v}_d^k) & \text{if } centered_{LL}(\boldsymbol{v}^i,\boldsymbol{v}^j,\boldsymbol{v}^k) \\ centered_{LR}(\boldsymbol{v}_d^i,\boldsymbol{v}_d^j,\boldsymbol{v}_d^k) & \text{if } centered_{LR}(\boldsymbol{v}_i,\boldsymbol{v}_j,\boldsymbol{v}_k) \\ centered_{RL}(\boldsymbol{v}_d^i,\boldsymbol{v}_d^j,\boldsymbol{v}_d^k) & \text{if } centered_{RL}(\boldsymbol{v}_i,\boldsymbol{v}_j,\boldsymbol{v}_k) \\ centered_{RR}(\boldsymbol{v}_d^i,\boldsymbol{v}_d^j,\boldsymbol{v}_d^k) & \text{if } centered_{RR}(\boldsymbol{v}_i,\boldsymbol{v}_j,\boldsymbol{v}_k) \end{array} \middle| \begin{array}{l} \forall i, j, k. \\ j \ne i \\ k \ne i \\ 1 \le i \le N \\ 0 \le j,k \le N \end{array} \right\}$$

$$centered_{LR}(v, v^i, v^j) \overset{\text{def}}{=} \Big((v.x_L + v.x_R)/2 = (v^i.x_L + v^j.x_R)/2\Big)$$

Preserve Aspect Ratio. The $\phi_{preserve\_aspect\_ratio}$ property ensures that the view aspect ratio (the ratio of width to the height) is preserved across all devices. However, this property applies only to those views which have one of the standard aspect ratios as defined below.

$$\phi_{preserve\_aspect\_ratio}(\boldsymbol{v}, \boldsymbol{v}_d) \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} ar(\boldsymbol{v}^i) = ar(\boldsymbol{v}_d^i) \text{ if } ar(\boldsymbol{v}^i) \in \{\frac{16}{9}, \frac{3}{2}, \frac{4}{3}, \frac{1}{1}, \frac{3}{4}, \frac{2}{3}\}$$

$$ar(v) \overset{\text{def}}{=} \frac{v.x_R - v.x_L}{v.x_B - v.x_T}$$

Pixel Perfect. We ensure that the synthesized constraints take into account the physical limitations of the device – the fact that device screens consist of a discrete number of pixels. To prevent rounding (which can introduce visual artefacts) we ensure that only solutions which do not require rounding are produced by restricting the handle points representation to be non-negative integers.

$$\phi_{pixel\_perfect}(\boldsymbol{v}_d) \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}_d|} \boldsymbol{v}_d^i.x_L \in \mathbb{Z}^{\geq 0} \wedge \boldsymbol{v}_d^i.x_R \in \mathbb{Z}^{\geq 0}$$

Inside Screen. The ϕ_{inside_screen} property checks that all views are rendered fully inside the device screen. This can be achieved by constraining each view handle point.

$$\phi_{inside\_screen}(d, \boldsymbol{v}_d) \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}_d|} \left(d.x_L \leq \boldsymbol{v}_d.x_L^i\right) \wedge \left(\boldsymbol{v}_d.x_R^i \leq d.x_R\right)$$

6.2 Incorporating User Feedback

As the synthesis problem is under specified, it is possible that even after satisfying all robustness properties, the synthesized layout is not the one the designer had in mind. It is therefore important that a designer can provide feedback and re-run the layout synthesis. As our synthesis algorithm is encoded as a logical formula, it naturally allows specifying a wide range of additional properties to be satisfied. However, instead of requiring the designer to write logical formulas we can simply render the layouts and allow the user to modify the absolute position and size of the rendered views. Then, the views which were changed are added as an additional input specification.

Summary. In this section we presented a synthesis algorithm that produces layouts that generalize well on multiple devices while crucially requiring an input specification only for a single device. To achieve this, the key idea is to design a set of properties satisfied by layouts that generalize well and encode them as a part of the synthesis task. Further, the properties are encoded in a modular way as logical formulas that allow the user to easily add more constraints if necessary.

7 SCALING LAYOUT SYNTHESIS WITH A PROBABILISTIC CONSTRAINTS MODEL

In this section we present our final version of the synthesis algorithm. We extend ψ_{multi_syn} in a way that results in: (i) a scalable algorithm that can synthesize real world layouts in less than a second, and (ii) an improved ability to generalize across multiple devices by taking into account constraints that are likely to be written by developers.

Key Challenge I: Scalability. As we will show in our evaluation, state-of-the-art SMT engines do not scale to solving the synthesis formula ψ_{multi_syn} directly. This is because the formula size is cubic in the number of views $\boldsymbol{v}$ and quickly becomes intractable to solve for all but very small sizes. The main scalability bottleneck is that both synthesis algorithms ψ_{single_syn} and ψ_{multi_syn} consider all admissible constraints for each view, out of which only two are synthesized – one horizontal and one vertical constraint. The key idea that enables us to make synthesis scale is that instead of considering all admissible constraints, we only consider those that are likely to make the synthesis formula satisfiable. That is, we are interested in learning a model $F\colon Constraint \to Constraint$ which takes as input all admissible constraints and returns only a subset of them. In the extreme case, F returns exactly the two constraints for each view that make the synthesis formula satisfiable, effectively solving the synthesis problem. Although creating such a perfect model is too expensive, we will show how to learn a model that is close to optimal (in our evaluation we return only 5 constraints) yet is very fast to compute. This allows us to significantly reduce the number of constraints considered by the synthesizer and efficiently generate complex real world layouts.

$$\psi_{multi\_syn+guided}(\rho \in View, \boldsymbol{d} \subset View, \boldsymbol{v} \subset View) = \max_{\sum_{i=1}^{|\boldsymbol{v}|} score^i} \psi_{multi\_syn}(\rho, \boldsymbol{d}, \boldsymbol{v})$$

$$\phi_{constraints} \overset{\text{def}}{=} \bigwedge_{i=1}^{|\boldsymbol{v}|} \left(\bigwedge_{k=0}^{|F(\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho))|} g_k^i \Rightarrow \left([\![\boldsymbol{c}_k^i]\!] \land score^i = P(\boldsymbol{c}_k^i, \boldsymbol{v}) \right) \right) \land g_0^i + \cdots + g^i_{|F(\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho))|} = 1$$

Fig. 12. Synthesis algorithm $\psi_{multi\_syn+guided}$ guided by a probabilistic model of constraints which selects constraints with the highest *score* that satisfy the synthesis formula ψ_{multi_syn}.

Key Challenge II: Natural Constraints. In Section 6 we defined a set of general properties which every layout should satisfy. However, for many designs, multiple options may be available and a designer has to select one of these based on their intentions. Although our model supports supplying additional constraints (described in Section 6.2) our goal is to reduce the amount of feedback the user needs to provide in order to synthesize the desired layout. For this purpose, we extend the synthesis algorithm such that if multiple constraints exists that satisfy the input specification it produces those that are more likely to be written by a developer.

Key Idea: Guiding the Synthesis via Probabilistic Model of Constraints. The key idea that addresses both issues, scalability and natural layouts, is to guide the synthesis with a probabilistic model of constraints $P\colon Constraint \rightarrow \mathcal{R}^{[0,1]}$ that assigns a valid probability to each constraint. Then, F is defined to simply return top K most likely constraints according to P. Our final synthesis algorithm $\psi_{multi\_syn+guided}$ is shown in Fig. 12. It extends ψ_{multi_syn} in two key aspects: (i) it considers only a subset of all admissible constraints ($F(\mathcal{C}(\boldsymbol{v}^i, \boldsymbol{v}, \rho))$), and (ii) out of those constraints $\boldsymbol{c}$ that satisfy the formula, it selects the ones which are most likely according to the probabilistic model P. In the remainder of this section we describe how to learn such a probabilistic model of constraints from a large dataset of layouts written by developers.

We now define a probabilistic model that assigns probabilities to constraints. A key challenge here is that, as illustrated in Section 3, constraint probability depends on the context. That is, the same constraints can have different probability depending on where the views they relate to are located on the screen. To solve this issue we learn the probabilistic model over a large dataset of layouts that capture the context in which constraints are written by developers.

Probabilistic Model Definition. Let $c \in Constraint$ be a relational constraint, $\boldsymbol{v} \subset View$ be a set of views and $\rho \in View$ be a content frame as defined in Section 4. We define the probability of a constraint as:

$$P(c, \rho, \boldsymbol{v}) = \frac{1}{Z(\rho, \boldsymbol{v})} \prod_{k=1}^{K} P_{f_k}(c \mid f_k(c, \boldsymbol{v}))^{w_k}$$

where $\{P_{f_k}\}_{k=1}^{K}$ is a set of probability distributions with associated weights $w_k \in \mathbb{R}^k$, $\{f_k\}_{k=1}^{K}$ is a set of feature functions and Z is a normalization function that ensures P is a valid probability distribution (where $\mathcal{C}(\boldsymbol{v}, \rho)$ denotes all admissible constraints defined over $\boldsymbol{v}$ and ρ):

$$Z(\rho, \boldsymbol{v}) = \sum_{c \in \mathcal{C}(\boldsymbol{v}, \rho)} \prod_{k=1}^{K} P_{f_k}(c \mid f_k(c, \boldsymbol{v}))^{w_k}$$

The intuition behind our definition is that the complex probability distribution of a constraint can be decomposed into a product of simpler probability distributions P_{f_k} over the set of views $\boldsymbol{v}$.

Table 2. Feature functions used to define a probabilistic model of constraints. We use $LineSeg(c.t_h, c.A, c.B)$ to denote the shortest line segment connecting handle points of views A and B related by constraint c.

Feature Functions: $f_k(c, \boldsymbol{v})$	Description
Margins	
$f_m \overset{\text{def}}{=} \langle orientation(c.t_h), c.m_L, c.m_R \rangle$	Returns the margins associated with the constraint c. Defines one model per orientation.
Bias	
$f_b \overset{\text{def}}{=} \langle orientation(c.t_h), c.b_h \rangle$	Returns the bias associated with the constraint c. Defines one model per orientation.
Distance	
$f_d \overset{\text{def}}{=} \langle class(c.t_h), \lvert\lvert LineSeg(c.t_h, c.A, c.B) \rvert\rvert \rangle$	Returns the shortest euclidean distance between A and B. Defines one model per constraint class.
Size	
$f_s \overset{\text{def}}{=} \langle orientation(c.t_h), s.t_h, \lfloor width/16 \rfloor \rangle$	Returns a tuple consisting of the view size type and the view width rounded to 16 pixels.
Orientation	
$f_o \overset{\text{def}}{=} \langle c.t_h, \text{arctan2}(LineSeg(c.t_h, c.A, c.B)) \rangle$	Returns the angle of the shortest line segment from A to B. Defines one model per constraint type.
Type	
$f_t \overset{\text{def}}{=} \langle c.t_h \rangle$	Returns the constraint type.
Intersection	
$f_i \overset{\text{def}}{=} \lvert\{v \mid \forall v \in \boldsymbol{v}.\ \texttt{intersects}(v, seg)\}\rvert$ where $seg = LineSeg(c.t_h, c.A, c.B)$	Returns the number of other views intersected by a line segment between h_s and h_t.

The goal of each P_{f_k} is to capture one relevant aspect that helps in deciding whether constraint c is likely or not. Although P_{f_k} and its feature function f_k can be complex, we show that even a small set of well designed simple functions is sufficient to capture the intuition behind good constraints. We note that even though the feature functions can depend on the context captured by the position of other views, they cannot condition on other constraints. This is important as it allows us to pre-compute all constraint probabilities before solving the synthesis formula.

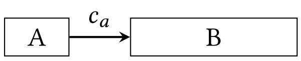
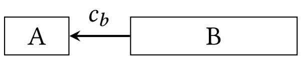
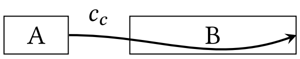
To estimate the distributions P_{f_k} we use a training dataset that reflects developer preferences instead of manually designing fixed distributions. For a given feature function f_k we estimate the constraint probability using maximum likelihood estimation (via counting):

$$P_{f_k}(c \mid f_k(c, \boldsymbol{v})) = \frac{1 + Count(f_k(c, \boldsymbol{v}))}{1 + |T| + \sum_{t \in T} Count(t)} \tag{1}$$

where $Count(f_k(c, \boldsymbol{v}))$ denotes the number of times the value computed by function f_k was seen in the training data, T denotes the range of f_k when evaluated on training dataset (i.e., set of all returned values) and $\sum_{t \in T} Count(t)$ is the number of training examples. To avoid returning zero probability for values not seen during training we use an approach called additive smoothing [Johnson 1932; Lidstone 1920] which adds one to the numerator and $|T|$ to denominator.

Feature Functions. The feature functions used in our work are formally defined in Table 2. Each feature function returns a tuple of values that are used to compute the constraint probability as defined by Equation 1. We use *orientation* and *class* helper function to return the constraint orientation (either horizontal or vertical) and class (relational, centering or circular) respectively. Note

Table 3. Values and probabilities computed by feature functions of different constraints relating two views.

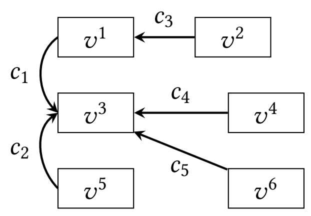
Constraint	Orientation		Intersection		Margins	
	$f_o(c, \boldsymbol{v})$	$P(c \mid f_o)$	$f_i(c, \boldsymbol{v})$	$P(c \mid f_i)$	$f_m(c, \boldsymbol{v})$	$P(c \mid f_m)$
	$\langle \mathcal{R}_{RL}, 0^\circ \rangle$	0.34	0	0.77	$\langle H, 10 \rangle$	0.06
	$\langle \mathcal{R}_{LR}, 180^\circ \rangle$	0.38	0	0.77	$\langle H, 10 \rangle$	0.06
	$\langle \mathcal{R}_{RR}, 0^\circ \rangle$	0.06	1	0.07	$\langle H, 60 \rangle$	0.005

that we use the constraint type (or its orientation/class) as part of the return value as a shorthand for defining a specialized probability distribution learned only from constraints of that type. Further, to prefer simpler constraints we use a regularization feature function which returns the number of unique constants and views used by a constraint c.

Example: Querying the Model. Consider two views A and B positioned next to each other and related with each other using a constraint as illustrated in Table 3. For each constraint, Table 3 shows the values computed by orientation, intersection and margins feature functions as well as their probabilities as computed by our model. The most likely constraint is c_b since the orientation 180° (i.e., right to left) is slightly more likely than 0° (i.e., left to right). This is due to the fact that the screen content tends to be written from left to right which results in right to left constraints (since we want to anchor the view to the left side). Note that the computed orientation probability for constraint c_c is significantly smaller than for c_a even though they both have angle value 0°. This is because for orientation we learn a separate model for each constraint type. As a result, we learn that aligning views with their right edges as done by constraint c_c is much more likely when the views are either above or below each other instead of side by side as in the above example. On the other hand, for margins we learn only two models, one for horizontal and one for vertical constraints. Therefore, since both constraint c_a and c_b in Table 3 are horizontal and have the same margin their probabilities according to the margin model are also the same.

Example: Training the Model. In Fig. 13 we show the training of the probabilistic model of constraints. Fig. 13 a) contains example of five constraints written by a developer that are used as a training dataset. For example, the constraint c_1 specifies that left edge of view v_1 is aligned to left edge of view v^3 with zero margin. For each of the constraints we first extract all the features, as shown in b), such as the orientation, margin or the type of the constraint. Based on the extracted feature we then define the probabilistic models using maximum likelihood estimation which first counts the number of times each feature appears in the training dataset and then uses Equation 1 to compute the constraint probability as shown in Fig. 13 c). For example, consider the probability of constraint c_3 according to the orientation model $P(c_3 \mid f_o(c_3, \boldsymbol{v})) = \frac{1+Count(f_o(c_3,\boldsymbol{v}))}{1+|T|+\sum_{t \in T} Count(t)}$. Here, $Count(f_o(c_3, \boldsymbol{v})) = Count(\langle \mathcal{R}_{LR}, 180^\circ \rangle) = 2$ is the number of times $\langle \mathcal{R}_{LR}, 180^\circ \rangle$ appears in the dataset which is twice (for constraints c_3 and c_4). $|T| = |\{\langle \mathcal{R}_{LR}, 180^\circ \rangle, \langle \mathcal{R}_{LR}, 160^\circ \rangle\}| = 2$ is the number of unique features seen during the training and $\sum_{t \in T} Count(t) = 3$ is the number of constraints in the model. Note that using the constraint type as part of the feature denotes a specialized probability distribution learned only from constraints of that type. Therefore, in Fig. 13 c) separate orientation models are defined for constraint types $\mathcal{R}_{LL}$ and $\mathcal{R}_{LR}$.

a) Training Dataset



b) Features Extracted from the Training Dataset in a)

Constraint	$f_o(c, \boldsymbol{v})$	$f_m(c, \boldsymbol{v})$	$f_t(c, \boldsymbol{v})$
c_1	$\langle \mathcal{R}_{LL}, 270°\rangle$	$\langle H, 0\rangle$	$\mathcal{R}_{LL}$
c_2	$\langle \mathcal{R}_{LL}, 90°\rangle$	$\langle H, 0\rangle$	$\mathcal{R}_{LL}$
c_3	$\langle \mathcal{R}_{LR}, 180°\rangle$	$\langle H, 10\rangle$	$\mathcal{R}_{LR}$
c_4	$\langle \mathcal{R}_{LR}, 180°\rangle$	$\langle H, 16\rangle$	$\mathcal{R}_{LR}$
c_5	$\langle \mathcal{R}_{LR}, 160°\rangle$	$\langle H, 16\rangle$	$\mathcal{R}_{LR}$

c) Probabilistic Constraint Models of the Training Dataset in a)

Orientation Model for $\mathcal{R}_{LL}$ Constraints

f_o	$Count(f_o)$	$P(c \mid f_o)$
$\langle \mathcal{R}_{LL}, 90°\rangle$	1	$\frac{1+1}{1+2+2} = 0.4$
$\langle \mathcal{R}_{LL}, 270°\rangle$	1	$\frac{1+1}{1+2+2} = 0.4$
Unseen	0	$\frac{1+0}{1+2+2} = 0.2$

Orientation Model for $\mathcal{R}_{LR}$ Constraints

f_o	$Count(f_o)$	$P(c \mid f_o)$
$\langle \mathcal{R}_{LR}, 180°\rangle$	2	$\frac{1+2}{1+2+3} = 0.5$
$\langle \mathcal{R}_{LR}, 160°\rangle$	1	$\frac{1+1}{1+2+3} = 0.3\bar{3}$
Unseen	0	$\frac{1+0}{1+2+3} = 0.1\bar{6}$

Margin Model for Horizontal Constraints

f_m	$Count(f_m)$	$P(c \mid f_m)$
$\langle H, 0\rangle$	2	$\frac{1+2}{1+3+5} = 0.3\bar{3}$
$\langle H, 16\rangle$	2	$\frac{1+2}{1+3+5} = 0.3\bar{3}$
$\langle H, 10\rangle$	1	$\frac{1+1}{1+3+5} = 0.2\bar{2}$
Unseen	0	$\frac{1+0}{1+3+5} = 0.1\bar{1}$

Type Model

f_t	$Count(f_t)$	$P(c \mid f_t)$
$\mathcal{R}_{LR}$	3	$\frac{1+3}{1+2+5} = 0.5$
$\mathcal{R}_{LL}$	2	$\frac{1+2}{1+2+5} = 0.375$
Unseen	0	$\frac{1+0}{1+2+5} = 0.125$

Fig. 13. Example of training a probabilistic model of constraints. For each constraint in a training dataset a) the features are extracted b) and used to train the probabilistic model c). The probabilistic models are trained using maximum likelihood estimation, that is, by counting how many times each feature appears in the training dataset. We use "Unseen" to denote the probability mass assigned to unseen features via smoothing.

Constraint Generation. Note that the probabilistic model can condition the constraint probability on the values of margins, bias or views it relates to. This is crucial for the model precision yet these are the values we are interested in synthesizing. To address this issue, we take advantage of the fact that constraints with one unknown variable can be resolved *locally* given the input specification $\boldsymbol{v}$ and the content frame ρ. For this purpose we instantiate all relative and circular positioning constraints for each pair of views $A, B \in \{v \cup \rho\}$ and solve for the margins and distance respectively. Further, we instantiate the centering constraints for each triple of views $A, B, C \in \{v \cup \rho\}$, by fixing the bias to be from a fixed set of values and restricting that either one of the margins is zero or they are both equal to each other. Finally, the logical formulas specified by the constraints are also simplified (e.g., by evaluating $\sin(\alpha)$ in circular constraints for known α). Using this approach allows us to score the constraints with the probabilistic model *before* solving the synthesis formula.

Summary. We have presented our approach to synthesizing relational constraints from examples. We started by introducing a new synthesis algorithm that solves the problem for a single device in Section 5. In Section 6 we extend the synthesis algorithm such that the synthesized layouts satisfy a set of robustness properties allowing them to generalize across multiple devices. Finally, in this section we show how to scale both algorithms to complex real world layouts using a probabilistic model of constraints to guide the synthesis towards satisfiable solution.

8 EVALUATION

In this section, we provide a thorough evaluation of our proposed approach implemented in a tool called InferUI that synthesizes Android layouts. We demonstrate that:

- *Scalability.* The synthesis algorithm scales to real world applications and synthesizes even the most complex layouts for a single device in $\approx$3 seconds.
- *Precision & Robustness.* The synthesized layouts based on a specification from only a *single* device generalize well to multiple devices with 92% of views being rendered at the screen location intended by the user.
- *Naturalness.* The synthesis algorithm creates natural constraints for developers: 62% of constraints it synthesizes match those written manually by developers.

We performed all experiments on a typical developer laptop with 2.40GHz Intel(R) Core(TM) i7-7560U CPU, running Ubuntu 16.04. To ensure correctness of our implementation we verify that the absolute view positions computed using the Android constraint layout solver version 1.0.2 and InferUI are the same for all synthesized layouts.

Dataset of Android Applications. For the purposes of evaluation we collected two datasets of real world layouts: (i) `PlayStore` dataset consisting of top 500 ranked applications on Google Play Store, and (ii) a `GitHub` dataset consisting of top 500 public repositories with the highest number of watchers on GitHub that contain `ConstraintLayout`. In order to compare to ground truth layouts written manually by developers we consider only layouts that use `ConstraintLayout`. Note that this does not limit the applicability of our approach since `ConstraintLayout` is the latest and most expressive layout available on Android. Further, we preprocess the dataset by removing incomplete layouts (e.g., position of some views is not constrained), layouts with invalid constraints (e.g., relating to a non-existed view) and layouts with circular constraints. Although the Android layout solver implementation is robust enough to render even such invalid layouts, we remove them as they are typically of low quality. Finally, we consider only layouts with at least two views.

Training a Probabilistic Model of Constraints. We trained our probabilistic model of constraints by extracting all user defined constraints from our datasets and evaluating them using the feature functions from Section 7. When evaluating applications from the `PlayStore` dataset we use the model trained on the `GitHub` dataset and conversely we use the model trained on the `GitHub` dataset when evaluating layouts from the `PlayStore` dataset. Because our model estimates constraint probability using maximum likelihood estimation (via counting) it is extremely fast to train and query – it takes less than a second to train models for both of our datasets.

Evaluated Algorithms. To evaluate the effectiveness of our approach we evaluate four algorithms:

- ψ_{single_syn} described in Section 5 that synthesizes layouts for a single device.
- ψ_{multi_syn} described in Section 6 that synthesizes layouts that generalize to multiple devices.
- $\psi_{multi\_syn+guided}$ described in Section 7 which is an extension of ψ_{multi_syn} that guides the synthesis with a learned probabilistic model of constraints.
- $\psi_{single\_syn+guided}$ which is a guided extension of the ψ_{single_syn} algorithm.

For all algorithms, we use the Z3 SMT solver version 4.6.0 [De Moura and Bjørner 2008] and set the timeout limit to one minute. Further, we guide the search for $\psi_{single\_syn+guided}$ and $\psi_{multi\_syn+guided}$ algorithms by selecting top 5 most likely constraints for each view. In case the problem is unsatisfiable we increase the number of selected constraints by 10 for each view in the unsat core. We repeat this process until a satisfiable solution is found or the time limit is reached. In our experiments top 5 constraints are sufficient in 69% of cases. They need to be expanded once, twice and three or more times in 21%, 7% and 3% of the cases, respectively.

Table 4. Average runtime (top) and percentage of successfully synthesized layouts (bottom) of increasing complexity of relational layout synthesis algorithms proposed in our work.

Synthesis Algorithm	Number of Views [2, 4)	[4, 8)	[8, 12)	[12, 16)	[16, 20)
Single Device					
ϕ_{syn}	29 ms	94 ms	490 ms	1.8 s	15s
$\phi_{syn+guided}$	37 ms	59 ms	129 ms	238 ms	519 ms
Multi Device					
ϕ_{multi_syn}	49 ms	580 ms	19 s	–	–
$\phi_{multi\_syn+guided}$	44 ms	95 ms	314 ms	3 s	3 s

Synthesis Algorithm	Number of Views [2, 4)	[4, 8)	[8, 12)	[12, 16)	[16, 20)	Synthesis Result SAT	UNSAT	TIMEOUT
Single Device								
ϕ_{syn}	99.0%	82.9%	38.3%	26.5%	16.6%	845	0	238
$\phi_{syn+guided}$	100%	100%	100%	100%	100%	1083	0	0
Multi Device								
ϕ_{multi_syn}	61.3%	23.5%	4.3%	0%	0%	327	9	747
$\phi_{multi\_syn+guided}$	98.7%	92.6%	73.0%	58.8%	41.7%	963	97	23

8.1 Scalability & Runtime

To evaluate the scalability of our algorithms, we synthesized layouts of increasing complexity. We consider layouts containing up to 20 views which includes 99.9% of layouts in our dataset. The average runtime of successfully synthesized layouts is shown in Table 4 (top). For a single device, the synthesis runtime is in milliseconds and even the most complex layouts are synthesized in a little over half a second. When synthesizing constraints for multiple devices, the runtimes are naturally higher but still very fast and less than 3 seconds. All runtimes reported are end-to-end, that is, including the time spent generating and scoring constraints using a probabilistic model.

In addition to runtime, we also evaluate the percentage of successfully synthesized layouts. The results are shown in Table 4 (bottom) together with a breakdown of why the synthesis was unsuccessful. The algorithm ψ_{single_syn} scales to layouts of size ≈ 10 after which it timeouts in 87.2% of cases. This is because the problem complexity growth is cubic in the number of views and quickly becomes intractable as more views are added. In contrast, the $\psi_{single\_syn+guided}$ succeeds for *all* layouts in our dataset. The problem of synthesizing layouts that generalize to multiple devices is much harder and can be solved directly by ψ_{multi_syn} only for the smallest of layouts containing less than 4 views. Guiding the synthesis using a probabilistic model significantly improves the scalability and allows us to synthesize up to three times larger layouts.

Note that when synthesizing layouts for multiple devices, in addition to timeout, the problem can also be unsatisfiable. For example, depending on the application design it is not always possible to fit all views into a smaller screen. Instead, the designer needs to create an alternative design that removes or restructures some of the views. In some cases however it is possible that the robustness properties are too restrictive and disallow valid layouts. For example, although preserving margins or centering is typically correct, some views might be centered simply by chance.

Table 5. Percentage of views that generalize to multiple devices. We consider a given view to generalize if its synthesized position is the same as the one specified by the user constraints.

	Percentage of Views that Generalize to Multiple Devices		
Metric	ψ_{single_syn}	$\psi_{single\_syn+guided}$	$\psi_{multi\_syn+guided}$
GitHub Dataset			
Horizontal Match	14.7%	78.3%	**89.1%**
Vertical Match	73.7%	88.3%	**96.5%**
Full Match	12.6%	69.4%	**86.5%**
PlayStore Dataset			
Horizontal Match	13.7%	85.8%	**93.4%**
Vertical Match	81.7%	92.4%	**98.9%**
Full Match	12.9%	75.5%	**92.3%**

8.2 Precision & Robustness

To evaluate the precision of our approach we compare the absolute view positions computed using our synthesized layout with the ground truth provided by the user (obtained by rendering layouts written by developers). Recall that the input to all of our synthesis algorithms is a set of absolute view positions on a single device. As a result, for a single device synthesis the precision is always 100% as we are guaranteed to satisfy the input specification. To evaluate the precision on multiple devices we synthesize layouts based on the specification for devices with screen size 360dp × 640dp[3] (e.g., Galaxy Nexus) and evaluate on devices with screen size in the range 341dp × 518dp to 384dp × 640dp. The results for both our datasets are shown in Table 5. We can see that ψ_{single_syn} generalizes poorly to only 12.6% and 12.9% of all views for the GitHub and PlayStore datasets respectively. This is expected as the synthesis only considers a single device for which the layout is synthesized. The $\psi_{single\_syn+guided}$ improves the generalization significantly by more that 55% for both datasets. Here, even though the synthesis algorithm still considers only a single device, the probabilistic model enables the synthesis to select constraints that are likely to generalize instead of any constraints that satisfy the input specification. Finally, we can see that $\psi_{multi\_syn+guided}$ leads to additional $\approx 15\%$ generalization improvement by using *both* the probabilistic model of constraints as well as considering multiple devices during synthesis.

Finding Layout Bugs in Existing Applications. The robustness properties defined in Section 6 can also be used to find layout bugs in existing applications. The results of evaluating robustness properties on both existing and synthesized layouts are shown in Table 6. We can see that although $\psi_{single\_syn+guided}$ significantly improves over ψ_{single_syn} it still violates at least one property in more than half of the analyzed layouts. On the other hand, the $\psi_{multi\_syn+guided}$ guarantees by design that all the properties are satisfied if the synthesis succeeds. Further, we have also found several property violations in existing applications. The most common violation is for $\phi_{pixel\_perfect}$ property which can cause small "off by one pixel" visual artefacts. More importantly, we also discovered serious issues that result in views being rendered outside of the screen or overlapping with each other. For concrete examples of bugs we found please refer to Fig. 11.

[3] dp stands for density independent pixels.

Table 6. Percentage of cases where synthesis algorithm violates robustness properties. For user defined layouts we also provide the total number of violations found.

	Percentage of Layouts that Violate Robustness Properties				
Property	ψ_{single_syn}	$\psi_{single\_syn+guided}$	$\psi_{multi\_syn+guided}$	**GitHub + PlayStore**	
$\neg\phi_{inside\_screen}$	86%	52.3%	0%	(4%)	21
$\neg\phi_{pixel\_perfect}$	40.3%	22.7%	0%	(8.5%)	41
$\neg\phi_{preserve\_aspect\_ratio}$	1.2%	0.6%	0%	(0.8%)	4
$\neg\phi_{preserve\_order}$	16.8%	24.5%	0%	(4%)	19
$\neg\phi_{preserve\_margins}$	85.0%	59.0%	0%	(5%)	24
$\neg\phi_{preserve\_centering}$	89.0%	55.2%	0%	(3.3%)	16

8.3 Synthesising Natural Layouts

We now evaluate the similarity of our synthesized constraints compared to those written manually by the users. It is a useful metric to optimize even though ideally the user never has to modify the constraints and in fact does not even need to be aware that they exists (especially if the user is not a developer but a designer). This is because, as illustrated in Table 5, synthesizing constraints that a user would write is an important indicator that the layout generalizes well. For our datasets we synthesize constraints that relate the same views as the user (centering constraints are considered to match only if both target views are the same) in 62% of the cases using the $\psi_{multi\_syn+guided}$ algorithm. Note that this percentage of constraints is significantly lower than the percentage of views that generalize well. This is because multiple constraints typically exists that all result in the same absolute position of the view. The ones that are finally selected depend on the preference of the developer such as writing constraints that relate views from left to right.

The Importance of Probabilistic Model of Constraints. We have already shown that guiding the synthesis using a probabilistic model of constraints is crucial for achieving scalability. For completeness we also evaluate the effect of phrasing the problem as maximum satisfiability, that is, returning the most likely constraints that satisfy the synthesis formula instead of returning any satisfying assignment. For $\psi_{single\_syn+guided}$, phrasing the problem as maximum satisfiability instead of satisfiability leads to 35% improvement in view generalization (Table 5) and 20% improvement in returning a constraint user would write. For $\psi_{multi\_syn+guided}$, although we did not observe an improvement in view generalization (it is already very high at 92%), the improvement in returning a constraint user would write is still 10%.

8.4 Incorporating User Feedback

So far all our experiments synthesized layouts from a single device input specification. However, as discussed in Section 6.2, our approach also supports refining the synthesized layout by extending the input specification with the user feedback. To evaluate such interactive setting, we performed a synthetic user study as follows: (i) synthesize layout using a single device input specification, (ii) render the layout on a set of devices, (iii) ask the user to randomly select a single view not rendered according to her design preferences. If such a view is found, we add it as part of the input specification and repeat the process from step (i). If no such view is found the synthesis terminates successfully. Note that this experiment is synthetic as we emulate the user using the constraints the developers wrote in our dataset of `GitHub` and `PlayStore` applications. Overall, using ψ_{multi_syn}

the user needs to provide 0, 1, 2 or 3 or more feedbacks in 63%, 25%, 8% and 4% of cases, respectively. That is, in 63% of cases the correct layout is synthesized without any user feedback and in 25% of cases the user needs to additionally modify the position of only a single view.

Summary. In this section we evaluated our system, called InferUI, and showed that it can synthesize layouts in 100% of cases when considering only a single device. Additionally, we successfully produce a single layout that generalizes to multiple devices in 89% of the cases. A key component that makes the synthesis scale to real world applications is a probabilistic model of constraints learned from a large set of layouts written by developers. We use the probabilistic model to efficiently prune the huge search space of possible layouts as well as to synthesize layouts likely to be written by a developer.

9 RELATED WORK

We next discuss some of the work that is most closely related to ours.

Layout Generation from Images. Works such as pix2code [Beltramelli 2017], REMAUI [Nguyen and Csallner 2015] and UI2Code [Chen et al. 2018] aim to generate layouts from images. Pix2code and UI2Code both use a language model based on deep neural networks which first encodes the input image using a convolutional neural network and then uses a recurrent neural network to generates a sequence of view names (e.g., `TextView`, `Button`, etc.) that are present on screen. In pix2code, the output sequence is represented in a domain specific language which encodes the simplest layout supported in Android (`LinearLayout`) and arranges all components either horizontally in a single column or vertically in a single row (or their combination). In UI2Code the output is a deeply-nested hierarchical structure with the names of all views present on the screen. This means that UI2Code does not in fact generate layouts but rather a sequence of components detected on the screen. As a result, a developer still has to write the layout manually and the output is only used to help in deciding which views are used in the screenshot. REMAUI uses OCR to identify text within an application screen. Identified words, detected edges and hand-engineered heuristics are used to segment the screenshot into user interface components, which are then exported into a layout. What layouts are supported as well as how the export is performed (the synthesis algorithm) is however not explained by the authors.

As can be seen, pix2code, UI2Code as REMAUI are all focused on the vision problem of identifying which views are present in the input image rather on the layout synthesis problem. In comparison, our work is the *first* to solve the complementary problem to view detection – once the views and their location are known, we synthesize a layout that *both* renders them on the screen according to this specification and generalizes to multiple devices.

Visual Errors Detection. In recent years, several techniques have been developed to detect visual errors that arise from cross-browser incompatibilities in web applications [Choudhary et al. 2012; Mahajan et al. 2016; Panchekha et al. 2018; Roy Choudhary et al. 2013] as well as in mobile applications [Moran et al. 2018]. For this purpose, a given application is typically first rendered on a set of devices (or web browsers) and searched for visual errors. If an error is found, an effort is made to localize its precise location (e.g., CSS property causing the error) which is then reported to the developer. In this line of work, the most recent and advanced tool called `Cassius` [Panchekha et al. 2018] formalizes a set of core components of CSS 2.1 standard and then verifies that a given web page conforms to 14 accessibility guidelines. In comparison, in our work we defined a set of robustness properties that can be verified in a similar way by formalizing the Android `ConstraintLayout`. More importantly, we developed a scalable synthesis algorithm that encodes the robustness properties as part of the specification, thus avoiding visual errors by construction at design time.

Program Synthesis. Combinations of program synthesis with images have been recently used to synthesize graphic programs from simple hand-drawn images [Ellis et al. 2017] as well as to infer program updates based for manipulating objects on a SVG canvas [Chugh et al. 2016]. In both of these approaches, the image is abstracted to a set of traces performed to draw the image. Although the visual output of these works and ours is similar (a set of rectangles drawn on a canvas) their internal representation is very different, which also requires developing different techniques to solve the task. In our case, the representation is declarative and based on a set of relational constraints compared to representing images as programs containing conditionals or even loops.

Furthermore, although our implementation currently returns a single most likely layout it might be useful to return multiple layouts amongst which the user can choose. To achieve this we could incorporate the techniques proposed by [Ellis et al. 2016] which allows efficient sampling of programs that satisfy given specification.

Machine Learning for Program Synthesis. Several approaches have been developed to accelerate program synthesizers by guiding the search towards a solution using a learned probabilistic model. Log-linear models were trained over a set of hand crafted features to guide synthesis of text processing task [Menon et al. 2013] and automatic patch generation [Long and Rinard 2016]. In [Balog et al. 2017] a neural network is trained to predict which predefined transformations are likely to be applied such that an input-output example is satisfied. In [Irving et al. 2016], a recurrent neural network is trained on a dataset of existing proof traces and used to improve the proof search of the theorem prover. Neural networks are also used by [Kalyan et al. 2018] for synthesis of text processing tasks which improves over prior work by combining statistical and symbolic search approaches and by not requiring hand crafted features. Finally, the work of [Lee et al. 2018] uses probabilistic higher order grammar [Bielik et al. 2016] that learns to guide A^* search to speed-up the synthesis in various domains including bitvectors, circuits and text processing tasks.

Compared to prior work, our work differs in three aspects – the domain over which the probabilistic model is learned, how the model is used to guide the synthesis and the type of the model. In our work, we introduce learning techniques to the domain of relational layout synthesis. The inputs over which our features are learned are a set of views positioned on a screen, instead of strings, numbers or proof traces. Next, our work considers the search procedure that solves the synthesis task to be a black-box – in our case an SMT solver. As a result, we guide the synthesis by restricting its search space, concretely, by selecting a subset of constraints that are extended if the synthesis fails. In contrast, prior works [Balog et al. 2017; Irving et al. 2016; Kalyan et al. 2018; Lee et al. 2018; Long and Rinard 2016; Menon et al. 2013] keep the search space unchanged and instead modify the search procedure used to find the solution. This is because while modifying the search procedure of A^* or breadth-first search considered in prior works is straightforward, it is challenging to modify the search procedure of the state-of-the-art SMT solvers that already contain number of carefully tuned heuristics and strategies that guide the search. Finally, the probabilistic model used in our work is maximum likelihood estimation that is extremely fast to both train and query. Here, the model precision can be further improved by using more complex models such as log-linear model, neural networks or probabilistic higher order grammar.

Learning from Big Code. Several works take advantage of the availability of large open source repositories and build tools that learn from big code, as surveyed here [Allamanis et al. 2018]. Similar to our work, these works learn a probabilistic model that is used to predict given properties (e.g., types [Katz et al. 2016; Raychev et al. 2015], variable names [Allamanis et al. 2015] or build probabilistic models of code [Bielik et al. 2016; Maddison and Tarlow 2014], etc.). Our work follows this line of work when defining the probabilistic model of constraints however with a different program representation consisting of views rendered on a device.

10 CONCLUSION

We presented a new approach for synthesizing relational layout constraints from examples and implemented it in a system called InferUI targeting Android applications. Our approach is based on a combination of techniques, enabling it to scale to complex real world layouts that generalize across multiple devices. Concretely, our algorithm synthesizes layouts with provable guarantees that satisfy both the input specification for a single device as well as a set of robustness properties (which aid in generalization). We showed that InferUI works well in practice and successfully scales to synthesizing complex layouts from top 500 ranked applications in the Google Play Store as well as top 500 most watched application on `GitHub`. Crucially, we achieved this without compromising on applicability – we support the latest `ConstraintLayout` used in Android and directly generate the corresponding source code to be used by the developer.

REFERENCES

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, New York, NY, USA, 38–49.

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (2018), 37 pages. https://doi.org/10.1145/3212695

Amigo. 2018. Amigo. https://play.google.com/store/apps/details?id=com.amigotrip.android

apptype. 2018. https://www.apptype.io/

Asobimasu. 2018. Asobimasu. https://github.com/DipanshKhandelwal/Asobimasu

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *5th International Conference on Learning Representations (ICLR '17)*.

Tony Beltramelli. 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. *CoRR* abs/1705.07962 (2017). arXiv:1705.07962 http://arxiv.org/abs/1705.07962

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of The 33rd International Conference on Machine Learning (ICML '16)*, Vol. 48. PMLR, New York, NY, USA, 2933–2942.

Candid. 2018. Candid. https://play.google.com/store/apps/details?id=in.voiceme.app.voiceme

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 (ICSE '18)*. ACM, New York, NY, USA, 665–676. https://doi.org/10.1145/3180155.3180240

Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2012. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proceedings of the IEEE 5th International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, Washington, DC, USA, 171–180. https://doi.org/10.1109/ICST.2012.97

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 341–354. https://doi.org/10.1145/2908080.2908103

Alex Corrado, Avery Lamp, Brendan Walsh, Edward Aryee, Erica Yuen, George Matthews, Jen Madiedo, Jeremie Laval, Luis Torres, Maddy Leger, Paris Hsu, Patrick Chen, Tim Rait, Seth Chong, Wjdan Alharthi, and Xiao Tu. 2018. Ink To Code. https://www.microsoft.com/en-us/garage/profiles/ink-to-code/

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2017. Learning to Infer Graphics Programs from Hand-Drawn Images. *CoRR* abs/1707.09627 (2017). arXiv:1707.09627 http://arxiv.org/abs/1707.09627

Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2016. Sampling for Bayesian Program Learning. In *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., Barcelona, Spain, 1297–1305. http://papers.nips.cc/paper/6082-sampling-for-bayesian-program-learning.pdf

FBook. 2018. FBook. https://play.google.com/store/apps/details?id=com.framgia.book

Ruozi Huang, Yonghao Long, and Xiangping Chen. 2016. Automatically Generating Web Page From A Mockup. In *The 28th International Conference on Software Engineering and Knowledge Engineering (SEKE '16)*. KSI Research Inc. and Knowledge Systems Institute Graduate School, Redwood City, San Francisco Bay, USA, 589–594. https://doi.org/10.18293/SEKE2016-231

Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Een, Francois Chollet, and Josef Urban. 2016. DeepMath - Deep Sequence Models for Premise Selection. In *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., Barcelona, Spain, 2235–2243. http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection.pdf

W. E. Johnson. 1932. Probability: The Deductive and Inductive Problems. *Mind* 41, 164 (1932), 409–423. http://www.jstor.org/stable/2250183

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations (ICLR '18)*.

Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating Types in Binaries Using Predictive Modeling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 313–326. https://doi.org/10.1145/2837614.2837674

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

George James Lidstone. 1920. Note on the general case of the bayes-laplace formula for inductive or a posteriori probabilities. *Transactions of the Faculty of Actuaries* 8 (1920), 182–192.

Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

Chris Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *Proceedings of the 31st International Conference on Machine Learning (ICML '14)*, Vol. 32-II. PMLR, New York, NY, USA, 649–657. http://dl.acm.org/citation.cfm?id=3044805.3044965

Sonal Mahajan, Bailan Li, Pooyan Behnamghader, and William G. J. Halfond. 2016. Using Visual Symptoms for Debugging Presentation Failures in Web Applications. In *Proceedings of the IEEE 9th International Conference on Software Testing, Verification and Validation (ICST '16)*. IEEE Computer Society, Washington, DC, USA, 191–201. https://doi.org/10.1109/ICST.2016.35

Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of The 30rd International Conference on Machine Learning (ICML '13)*, Vol. 28-I. PMLR, New York, NY, USA, 187–195.

Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated Reporting of GUI Design Violations for Mobile Apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 165–175. https://doi.org/10.1145/3180155.3180246

Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *Proceedings of the 30th ACM/IEEE International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 248–259. https://doi.org/10.1109/ASE.2015.32

Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying That Web Pages Have Accessible Layout. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3192366.3192407

psd2android. 2018. http://www.psd2androidxml.com/

psd2mobi. 2018. https://www.psd2mobi.com/service/psd-to-android-ui

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

Steven P. Reiss. 2014. Seeking the User Interface. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 103–114. https://doi.org/10.1145/2642937.2642976

replia. 2018. http://www.replia.io/

Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2013. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 702–711. http://dl.acm.org/citation.cfm?id=2486788.2486881

Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Andrew J. Ko. 2018. Rewire: Interface Design Assistance from Examples. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 504, 12 pages. https://doi.org/10.1145/3173574.3174078

Clemens Zeidler, Christof Lutteroth, Wolfgang Stuerzlinger, and Gerald Weber. 2013a. Evaluating Direct Manipulation Operations for Constraint-Based Layout. In *14th IFIP Conference on Human-Computer Interaction (INTERACT '13)*. Springer, Berlin, Heidelberg, 513–529. https://doi.org/10.1007/978-3-642-40480-1_35

Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. 2013b. The Auckland Layout Editor: An Improved GUI Layout Specification Process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 343–352. https://doi.org/10.1145/2501988.2502007

Clemens Zeidler, Gerald Weber, Wolfgang Stuerzlinger, and Christof Lutteroth. 2017. Automatic Generation of User Interface Layouts for Alternative Screen Orientations. In *16th IFIP Conference on Human-Computer Interaction (INTERACT '17)*. Springer, Berlin, Heidelberg, 13–35. https://doi.org/10.1007/978-3-319-67744-6_2