Synthesize iOS Device Layout Code in SwiftUI

DAVIN TJONG, UCSD, United States
RUOHAN HU, UCSD, United States
QIYUE WANG, UCSD, United States

In their daily work, mobile/web developers often need to replicate a UI design given to them with some UI layout code, such that the UI generated by the code is exactly the same as the design and can generalize across all different screen sizes. Many human errors can arise in this process. For example, the developer might forget some necessary padding or margin specifications. We have developed a UI code synthesis tool for SwiftUI, called SwynthUI, where the user can specify the layout of UI elements by drawing rectangles on a canvas. We believe this tool can make the process of writing layout code in SwiftUI easier.

ACM Reference Format:
Davin Tjong, Ruohan Hu, and Qiyue Wang. 2021. Synthesize iOS Device Layout Code in SwiftUI. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (March 2021), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Brief Description of Tool

Our synthesis tool provides an canvas board for users to draw UI elements in the form of rectangles. After cleaning the inputs (changing the size and position of those elements), the tool will input the coordinates along with constraints into a SMT constraint solver. And after getting the result from the solver, the tool will generate SwiftUI code and present it to the user.

1.2 Motivation

As discussed in the paper applying synthesis to Android's ConstraintLayout [Bielik et al. 2018], designing layouts and implementing them involves many common bugs and inconveniences. Often, the programmer must take the design created by the designer and make sure that it works on different screen sizes, leading to many common generalization errors. We employ similar techniques for the domain of SwiftUI, both because no such tool exists for SwiftUI that we have found, and because the semantics of SwiftUI make it conducive to a user-facing synthesizer. Also, we have not seen work that synthesizes based on messy user input, which would make these tools much easier to use.

1.3 Goals

The goal of the tool is to generate SwiftUI code that matches the user intention as much as possible. This means that we need to infer both the hierarchy of the views the user has drawn, but also that we need to guess exact values that match what the user intended to input. In addition, the code has to be readable so that even if it does not work properly, it should be easy for users to modify.

1.4 SwiftUI as a Synthesis Domain

There are several properties of SwiftUI that makes it a good domain for our tool. The semantics of SwiftUI simplify the problem of infering hierarchy because the hierarchy in SwiftUI is limited primarily to HStacks and VStacks, which are horizontal and vertical collections of views, respectively.

Authors' addresses: Davin Tjong, UCSD, United States, davintjong@gmail.com; Ruohan Hu, UCSD, United States, r8hu@ucsd.edu; Qiyue Wang, UCSD, United States, qiw131@ucsd.edu.

2021. 2475-1421/2021/3-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This makes infering the hierarchy much simpler, as we discuss later. In addition, our layout code only needs to be generalizable within iPhone screen dimensions, which also makes it easier to generate good layout code. Finally, the abstraction of a View in the language makes it easy for the user to change views in the synthesized layout to whatever they want.

2 USER INTERACTION

We have a few UI goals on this project: The first is that the user should be able to drag and drop without worrying to much about being precise. Another priority is for the user not to have to understand how SwiftUI hierarchy is structured in order to user it. These two goals are achieved by inferring what the user intended to input, both in terms of precision and the hierarchy.

We'll now start an example that well will use throughout the report to explain how the synthesizer works. For now, we will just explain the workflow from the user's point of view. We will demonstrate using the tool to synthesize SwiftUI for the following layout (Fig. 1).

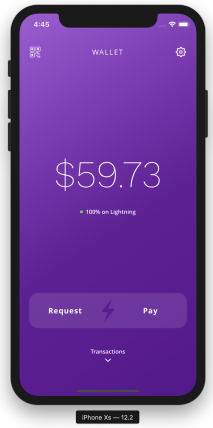

Fig. 1. Layout that we will synthesize

Upon opening the tool, the user sees a blank canvas (Fig. 2). To synthesize the layout, the user draws rectangles by dragging and dropping black rectangles where they want their views. In this case, rectangles are drawn for text, buttons, and symbols.

In addition, they can press the 'framed' button, which toggles the mode for drawing rectangles. Instead of black boxes, the drawn rectangles are outlined instead. This is a means to allow the user to tell the synthesizer to try to make it so that these views are unframed, meaning that they will expand with the size of the screen. In the worked example, the user wants the text boxes that contain the wallet total to be at the center of the screen, regardless of the screen size. So, the user draws unframed views above and below them, with the intention to replace them with spacers later. After drawing all the views, the user presses the 'snap' button to clean their input for error, which aligns the drawn views to more sensible values. 'snap' rearranges the views on screen, and if the user is satisfied with the output, they then press 'submit', which runs the synthesizer and copies the synthesized code to the clipboard. The user then takes the copied code and pastes it into XCode, where they see their layout in the preview (Fig. 3).

The user now changes the 'Color.black' to be buttons, text, spacers, or whatever they choose.

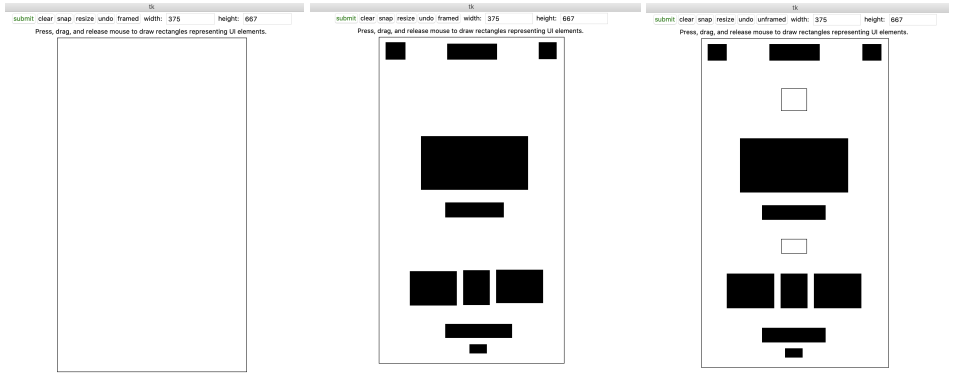

Fig. 2. (Left) Starting screen. (Middle) After drawing framed views. (Right) After drawing all views and pressing 'snap'.

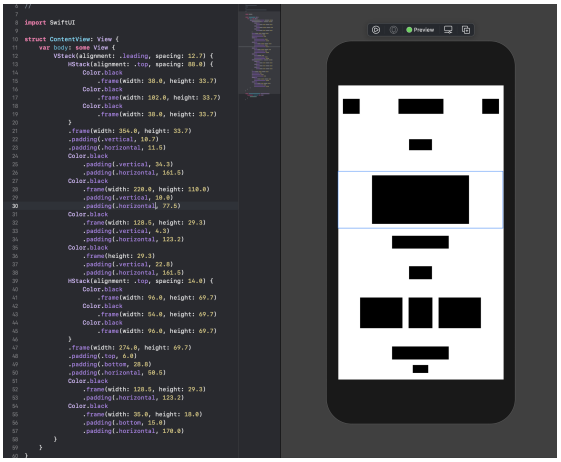

Fig. 3. The initial result of the synthesis.

The results can be seen in Figure 4. Note that the synthesis was done on the dimensions of the iPhone 8, however this can be changed in the tool using the 'resize' button. Figure 4 also shows how the layout generalized correctly to keep the wallet value at the center of the screen.

3 LAYOUT SYNTHESIS

The synthesis algorithm works as follows: first, we infer hierarchy from the input using a simple algorithm. Then, using the hierarchy structure, we 'snap' the user's input such that similar values for width, height, and spacing within a hierarchy are made to be equal, and center the views along the minor axis, if possible. Finally, we use z3's optimizer to solve for constraints hierarchy-by-hierarchy, and turn those constraints into SwiftUI code.

3.1 Hierarchy Inference

The algorithm we use the infer hierarchy takes inspiration from scheduling algorithms. The source code can be found here.

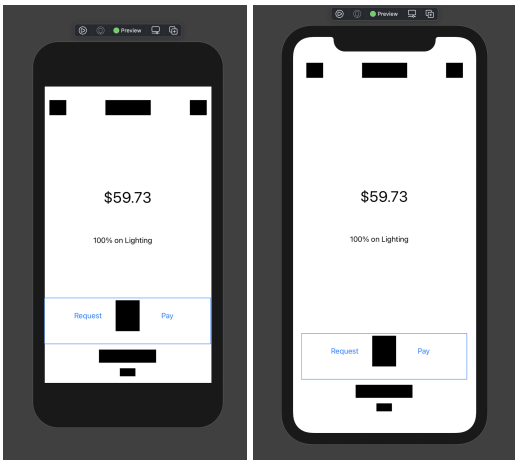

Fig. 4. (left) The synthesis result on an iPhone 8. (right) the synthesis result on an iPhone 11

We first pick an axis (we will try both axes to start with and use whichever leads to the simplest hierarchy). The algorithm wants to split the views into the smallest number of groups possible such that no two groups have any intersecting views along the axis, and within each group there are no gaps along the axis. To do this, we first sort the views along the axis in order of increasing end values. Then, we start a group by popping the first element, and while there is an element in the list with a start value that is earlier than the current group's end value, we pop the first element and add it to the list. We repeat until the list of views is empty. Then, for each loop, we recursively repeat the algorithm with the axis flipped.

To work through the algorithm on the worked example, pick the vertical axis, and the views are split up as shown.

Fig. 5. The views along the vertical axis split into groups.

Note that the algorithm tries both horizontal and vertical axes to start, and the way it decides which one is better is in terms of complexity, defined as the number of times the algorithm was

called (or, the number of non-leaf nodes in the hierarchy). In this case, the vertical axis is chosen because it leads to 3 non-leaf nodes in the hierarchy, whereas starting with the horizontal axis would cause 4 non-leaf nodes in the hierarchy.

It is also important to note that non-leaf nodes (beside the root) are assumed to have the smallest dimensions possible such that they encapsulate their children. This is part of the algorithm primarily for ease of implementation, as it would be better to infer the size of the view such that it both encapsulates its children and fits well within its parent. This issue could also be addressed by simply allowing the user to draw views within each other to dictate how the hierarchy is structured, which would be trivial to implement.

3.2 Snapping

After the hierarchy is inferred from the messy input, we use a pretty naive method for cleaning the input. The algorithm simply bins views within the hierarchy in terms of both height and width, and takes views within the bins and averages the values out. While this works reasonably well, there are several disadvantages to this approach and it is a major area for improvement for the tool.

Fig. 6. The top bar of the worked example, pre-cleaning.

Let us take the top bar of the worked example: For each axis, we bin along the size. For the vertical axis, they all have similar heights, so they all get put in one bin and averaged. However, for the horizontal axis, two are similar size, but one isn't, so they get put into separate bins. For spacing, the spacing between the views is about even, so they get binned together as well and averaged.

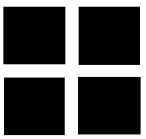

Fig. 7. A 2x2 grid

An obvious flaw with this method is that it doesn't account for views in other hierarchies. Thus, in a 2x2 grid as shown above, the tool wouldn't be able to bin views across two of the VStacks (or HStacks), and the views in the grid wouldn't get set to the same size.

A way to improve the the snapping would be to not only take into account all views at once, but work with the constraint solver to move views such that the solver can generate constraints with the least cost. Another possibility would be to use a general linear model like is done in Mockdown, a tool for synthesizing web layouts [Anonymous 2021].

3.3 Constraint Solver

Our constraint solver simply uses z3's optimizer. We find closed forms for the dimensions of all of the views within the hierarchy in terms of SwiftUI constraints, and solve for those constraints, maximizing a few values which we think lead to more plausible programs.

Because we already inferred the hierarchy of the program, and all children of the views are nicely aligned along their axis, it is quite simple to encode SwiftUI code as simple constraint values. The values being: Spacing, Alignment, Padding in each direction for each view, and Frame (size) in each dimension along each view. Of course, there are additional complications to this, being that programmers can leave the frame of the view undefined, in which case it will take up as much space as possible, split among unframed views. For both padding and frames, if the value is 0, it is treated as undefined. Also, due to how the problem is set up, the problem should always be satisfiable. The code defining the constraints are listed here.

In addition to solving for these constraints, we also optimize the number of frames in the hierarchy that match, symmetric padding, and we prefer if the alignment is centered. Also, we optimize to have the most spacing possible, as we would rather have higher spacing then it to have padding individually defined for each view.

As mentioned earlier, we also allow the user to annotate views to be framed or unframed. This is handled in the solver by adding constraints to framed views that require them to have nonzero frames. In the case of unframed views, we do not add constraints, rather we tell the solver to maximize the number of views that the user annotated as unframed that are indeed unframed. The reason we do this is because it is that unframed views in the same hierarchy must have the same size, so requiring the views to be unframed may make the problem unsatisfiable.

For the working example, there are three different non-leaf nodes in the hierarchy, meaning that we will run the solver three times in total. Running the solver on the hierarchy from Fig. 6 yields the following constraints:

$$
\begin{aligned}
&Alignment = 0, Spacing = 88 \\
&PadTop0 = 0, PadBot0 = 0, PadLeft0 = 0, PadRight0 = 0 \\
&FrameHeight0 = 16833333333333331/500000000000000, FrameWidth0 = 38 \\
&PadTop1 = 0, PadBot1 = 0, PadLeft1 = 0, PadRight1 = 0 \\
&FrameHeight1 = 16833333333333331/500000000000000, FrameWidth1 = 102 \\
&PadTop2 = 0, PadBot2 = 0, PadLeft2 = 0, PadRight2 = 0 \\
&FrameHeight2 = 16833333333333331/500000000000000, FrameWidth2 = 38
\end{aligned}
$$

This gets translated into SwiftUI as follows:

```
HStack(alignment: .top, spacing: 88.0) {
    Color.black
        .frame(width: 38.0, height: 33.7)
    Color.black
        .frame(width: 102.0, height: 33.7)
    Color.black
        .frame(width: 38.0, height: 33.7)
}
```

Fig. 8. The synthesized code.

Of course, if any of the children weren't leaf nodes, the 'Color.black' would be replaced by a VStack, with the results of the solver on that node.

4 EVALUATION

We would have preferred to do a user study, but using the tool on a few example apps is enough to show some of the strengths and weaknesses of the tool. Below are some sample screens that we mocked up in our tool, then showed how it displayed on an iPhone 8, which is the same dimensions as the tool synthesized in, and how it generalizes to the iPhone 11, which is significantly larger.

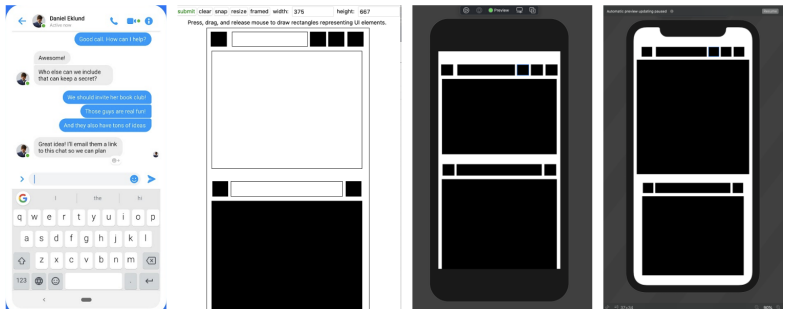

Fig. 9. Evaluation on Facebook Messenger.

On Facebook Messenger, it is able to successfully synthesize reasonable code, however, because the user isn't able to differentiate between having the width/height be framed/unframed, the synthesized layout isn't able to generalize as well as we would have liked. You can see in the keyboard that it doesn't expand horizontally in the iPhone 11.

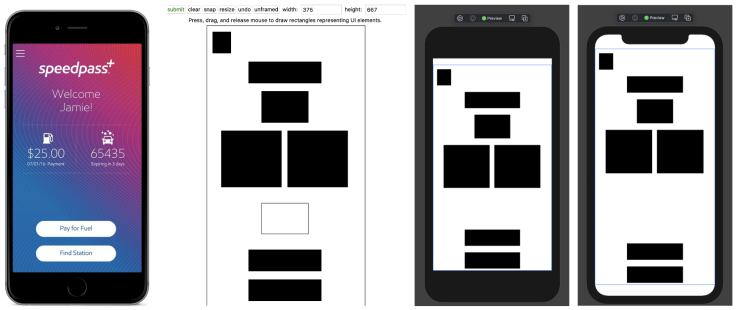

Fig. 10. Evaluation on Speedpass.

On Speedpass, the synthesizer does a good job of generalizing, as this specific home screen doesn't pose any challenges that are hard for the synthesizer.

Fig. 11. Evaluation on Canvas.

On Canvas, we can see that the width of the bottom toolbar does generalize correctly to the larger screen.

In terms of speed, the synthesis takes a fraction of a second, so we didn't feel the need to time it since it is more useful to evaluate its correctness. However, it is important to note that due to what is likely implementation errors, there are certain situations where the tool will hang when you attempt to synthesize. That said, we haven't found a way to reproduce it, partly because it happens so rarely.

5 CONCLUSION

Our tool, SwynthUI, allows users to write generalizable SwiftUI code without having to understand how SwiftUI positions elements or organizes the hierarchy. It is fast and reasonably easy to use, though it has a few weaknesses.

Currently, the way the tool cleans user inputs is relatively naive. A better solution would either involve working with the constraint solver or by using a generalized linear model as is done in Mockdown [Anonymous 2021]. In addition, the code produced, while accurate, will often include a large amount of constants for padding, spacing, and frames. This isn't much of a problem from the user's perspective, as they can ignore these and simply replace the views with their own. Rather, this is simply due to the fact that allowing the user to draw freely doesn't usually result in the same SwiftUI layouts a programmer would create without the synthesizer.

Extending the tool to allow users to specify hierarchy by drawing nested views to specify if only the width or only the height should be framed would not only be trivial, but greatly increase the quality of code produced.

REFERENCES

Anonymous. 2021. Synthesis of Web Layouts from Examples. (2021). Provided by Nadia Polikarpova.

Pavol Bielik, Marc Fischer, and Martin Vechev. 2018. Robust Relational Layout Synthesis from Examples for Android. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 156–184.