

On a Declarative Guideline-Directed UI Layout Synthesis

Dmitrii Kosarev

Dmitrii.Kosarev@pm.me

Saint Petersburg State University, Russia

Denis Fokin

denis.fokin@gmail.com

Saint Petersburg, Russia

Peter Lozov

lozov.peter@gmail.com

Saint Petersburg State University, Russia

Dmitry Boulytchev

dboulytchev@math.spbu.ru

Saint Petersburg State University, Russia

Abstract

We address the problem of completely automatic and declarative way to laying out the elements of user interface in a way prescribed by a set of designer-specified guidelines. We present a model of UI which incorporates all relevant notions and features, and describe an approach to automatic UI layout synthesis based on utilization of relational verifiers. We discuss how the techniques of relational programming fit for solving this problem and showcase the (preliminary) results of the evaluation of the approach we suggest.

ACM Reference Format:

Dmitrii Kosarev, Peter Lozov, Denis Fokin, and Dmitry Boulytchev. 2022. On a Declarative Guideline-Directed UI Layout Synthesis. In *Proceedings of The Fourth miniKanren and Relational Programming Workshop (miniKanren 2022)*. ACM, New York, NY, USA, Article 1, 11 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

The user interface (UI) is the way through which a user interacts with a device, application, or a website. There are several approaches to provide such interaction, e.g. GUI (graphical user interface), CLI (command line interface), VUI (voice user interface), and so on. GUI is the most popular among others because of a low entry threshold for a user. Despite the low entry threshold, GUI is rather a complicated thing to design [13] because it must be intuitive, concise, useful and reachable (user should easily access UI controls, e.g. with a thumb on mobile screen [1]). At the same time GUI is a

visual representation of controls and its design requires understanding of how to acquire the aforementioned properties through visual aids.

In GUI applications UI is a set of components that provide information and controls for the user to accomplish specific tasks with the interactive system [15]. GUI design has to consider not only UI but UX (user experience) which results from the user's internal and physical state, prior experiences, attitudes, skills, abilities and personality, and from the context of use [15].

Modern GUI applications may follow different paradigms, e.g. MVC (Model-View-Controller) [20]. But every approach tends to introduce a separation between business logic and its visual representation [3]. The latest approaches are declarative and the view layer of MVC may be extracted into independent libraries [8].

Communication and iterations are the major problems in GUI development [12]. In order to minimize iterations, the GUI development can be done with UI design tools. Modern UI design tools utilization provides design speed increase, simplification of changes, and results in uniform design [22]. Some of the tools, like Figma, allow to create GUI from a prototype [4].

Another approach is to formally describe design principles as *guidelines*. Design guidelines define style, layout, components, text, and accessibility [10]. The formalization allows designer to achieve acceptable results just by following the rules. The guidelines are usually provided by:

1. operating system (macOS, Windows);
2. desktop environment (KDE, GNOME);
3. platform or framework [23];
4. development team.

Customer requirements could be considered as guidelines as well.

Existing approaches [9, 14, 21] do not allow to take into account all these requirements automatically. As a result, guidelines must be followed manually by designers and developers, hence designing a user interface becomes a complex and error-prone task.

In this paper we present a framework to synthesize the layouts for UI elements with respect to rules defined in the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

miniKanren 2022, September 15, 2022, Ljubljana, Slovenia

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00

<https://doi.org/XXXXXXX.XXXXXXX>

design guidelines. We give a formal model that allows to specify guidelines and to describe the logical structure of the UI. We also devise a completely automatic approach, based on relational programming [11], to synthesize layouts in terms of layout constraints. These constraints, in turn, are solved using Z3 solver [7], providing the absolute coordinates for all UI controls.

Throughout the paper we use OCAML [17] as functional implementation language and OCANREN [16] as relational language implementation.

2 UI Model

In this section we present a certain model which incorporates all the important features of the domain and specifies the problem we are dealing with in a precise form.

We can identify two important aspects of the UI: *structure* and *layout*. *Structure* describes the set of UI elements and their relations; *layout*, on the other hand, determines their relative placement. In our model, the structure is completely invariant w.r.t. the layout: regardless what a layout could be the assortment of the elements and their logical/functional dependencies remain the same.

In our approach we consider structure as a set of named *relations* between elements. For example, let us have the following UI form (see Fig. 1).

We can describe its structure as follows:

- there are three UI elements:
 1. a check box;
 2. a text label;
 3. a drop-down list (combobox).
- there is a dependency relation between the text label and the combobox since the former *describes* the latter; thus, the text label and the combobox constitute a compositional entity;
- there is an order relation between the checkbox and this compositional entity since (as we speculate) there is an implied dependency caused by the importance of the components from the perspective of the UI author.

In abstract terms, we can depicture this structure as follows (see Fig. 2).

For a given structure its *layout* can be specified using a set of primitives describing the placement of elements, their alignment and other similar properties. In the given example the label and combobox are layed out horizontally next to each other with a certain horizontal inset, the checkbox is stacked over the compositional label-combobox pair with a certain vertical inset, and the whole layout is left-justified. We can depicture the layout in question as follows (see Fig. 3).

Finally, we treat UI *guidelines* as mappings which (conditionally) relate structure to layout. Essentially, guidelines prescribe what layout primitives should be used if certain UI elements are connected by certain relations in the structure.

In the following subsections we uncover more details of the notions introduced above.

2.1 UI Structure

As we observed above, the UI structure is comprised of a set of UI elements and a number of named relations between them. The set of elements incorporates all conventional UI controls used in practice: text labels and textboxes, comboboxes, radio buttons, checkboxes, etc. The concrete nomenclature of these elements is mostly irrelevant for our approach as we consider them as fully abstract items.

For now we distinguish the following types of relations between UI elements:

- *Composition*: a number of elements can be composed together; their composition is considered as an extra, *virtual* element, which can be related to other elements as well.
- *Description*: one element can be considered as a description for another (for example, a text label or a check box for another element).
- *Subordination*: one element can be considered as a “subordinate” of another (for example, a text field, enabled by a checkbox).
- *Ordering*: elements can be ordered. The ordering may be used to reflect various *ad-hoc* intentions like the flow of data or operations, alphabetic ordering, etc.
- *Grouping*: elements can be grouped together. This relation is similar to the ordering, but has a weaker semantics. For example, it might be possible to permute the elements within a group or intersperse them with other elements.
- *Properties*: a number of one-to-one relations between elements and other data domains like strings, numbers, etc. These relations are used to uniformly encode various element properties essential for layout (or concrete layout guidelines) like text, sizes, etc.

As for now we consider connected structures only. In other words, each element is related to some other. If the structure can be divided into several unrelated parts, each of them will be solved independently with possible overlap.

Reiterating the example in the Fig. 1, we can now provide a more concrete description of its UI structure (see Fig. 4).

From the UI structure point of view these relations are considered as fully abstract entities. Their semantics and impact on the layout is determined by the layout guidelines only.

2.2 UI Layout

We have identified the following set of layout primitives:

- vertical composition of elements (element C_1 directly on top of the element C_2):

vert (C_1, C_2)

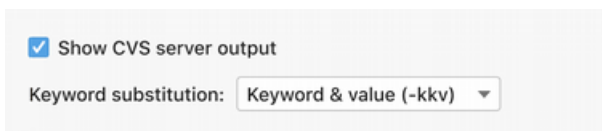


Figure 1. A Sample UI Form

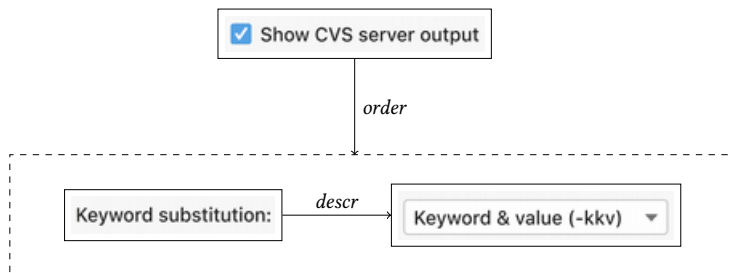


Figure 2. Sample UI Form Structure

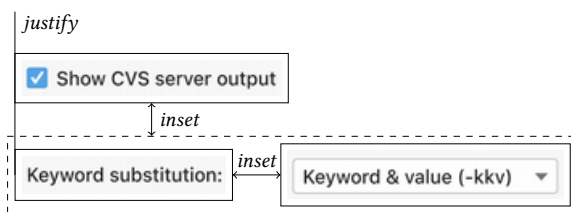


Figure 3. Sample UI Form Layout

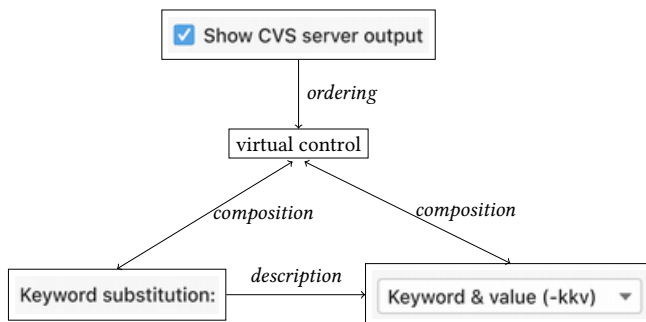


Figure 4. Sample UI Form Concrete Structure

- horizontal composition of elements (element C_1 directly on the left of the element C_2):

hor (C_1, C_2)

- vertical alignment of elements (does not affect the horizontal position of elements):

valign (C_1, C_2)

- horizontal alignment of elements (does not affect the vertical position of elements):

halign (C_1, C_2)

- horizontal indentation of one element to another:

indent (C_1, C_2)

This is a somewhat simplified model; in reality the additional arguments should be specified for these primitives which describe the *insets* and *alignments* of the elements being layed out. An inset specifies a desirable vertical or horizontal space between elements; as for alignments, different kinds of those are distinguished: top, center, bottom, and baseline vertical alignments, and left, right, and center horizontal alignments. As for now we consider the insets being a predefined constants and limit the alignments to some

default ones; we argue that this choice does not undermine the approach we describe.

This primitives can be orthogonally used for different elements: for example, elements A and B can be composed vertically, B and C – horizontally, and A and C can be horizontally aligned by center, which would provide the following layout (see Fig. 5). Note, the constraints in this example do not require B and C to be aligned *vertically*.

2.3 UI Guidelines

UI guidelines are a system that maps relations between elements to layout primitives. It can be generically described as a system of rules in the form

$$\mathfrak{P}[\mathfrak{C}] \rightarrow \mathfrak{L}$$

where

- \mathfrak{P} is a *relational pattern*. A pattern tests if a UI structure being layed out contains elements satisfying relations specified in the pattern. Those elements can be bound to the variables in the pattern.
- \mathfrak{C} is a *constraint*. A constraint can be used to provide additional checks not incorporated into a pattern (for example, it can put some extra restrictions on the sizes of elements and other their properties, such as contents of labels, etc.)
- \mathfrak{L} is a composition of layout primitives, which can contain occurrences of pattern variables.

The structure being layed out according to the UI guidelines is matched against the patterns, and, if a match succeeds and a constraint is satisfied, then the corresponding layout primitive instances are added to the set of the layout primitives collected so far. Unlike conventional pattern-matching in mainstream languages, a few guideline rules can be applied simultaneously (*nondeterministic* matching).

One basic form of a relational pattern is a relation name and two variable occurrences:

$$X \xrightarrow{rel} Y$$

A UI structure satisfies this pattern iff there are two (not necessarily distinct) elements X and Y connected by the relation rel .

Another basic form is a *property pattern* in the form

$$rel(X, p)$$

where rel is a name for property relation, X is a variable, and p is a property value. A UI structure satisfies this pattern iff there is an element X whose property rel has the value p .

We also consider composite patterns in the form $\mathfrak{P}_1, \dots, \mathfrak{P}_k$ where \mathfrak{P}_i are regular patterns; we treat the semantics of the composition as a conjunction of semantics for individual patterns (*and*-pattern). In principle, *or*-patterns and even

not-patterns can be also defined, but for now we did not discover the use cases for them in the existing guidelines.

As an example consider an encoding for a fragment of JetBrains UI guidelines [24] (see Fig. 6). In the table the left column shows the encoding of guidelines, while the right one – their informal presentation as given by JetBrains. The constant K in the second section of the table is introduced to express the informal condition “an input box is long, and the horizontal space is limited”. Its actual value may depend on the size of the enclosing pane, screen resolution, etc.

3 Synthesizing Layout Constraints

The first task in guideline-based layout synthesis is generation of layout primitives for a given structure which respect the guidelines in question. Note, for given guidelines and structure there can be potentially multiple admissible layouts. We solve this problem by utilizing *relational verifier*.

A verifier is a procedure which for a given guidelines \mathcal{G} , UI structure \mathcal{S} and a set of layout primitives \mathcal{L} checks if this layout for this structure is admissible w.r.t. these guidelines:

$$\text{verify}(\mathcal{G}, \mathcal{S}, \mathcal{L}) = \begin{cases} \text{true} & , \mathcal{L} \text{ respects } \mathcal{G} \\ \text{false} & , \text{otherwise} \end{cases}$$

This verifier, being converted into a relational form verify^o , can be used as a layout *synthesizer* [18]:

$$\text{synth}(\mathcal{G}, \mathcal{S}) = \text{run}^\alpha [\text{verify}^o(\mathcal{G}, \mathcal{S}, \alpha, \text{true})]$$

Here $\text{run}^\alpha [g]$ is a conventional MINIKANREN primitive for searching for all values of a variable α which make the goal g to succeed; it returns a (potentially infinite) stream of these values. Thus, the problem of layout constraints synthesis can be reduced to the problem of (relational) verifier construction.

In our approach we provide a specialized version $\text{verify}^o_{\mathcal{G}}$ for every set of guidelines of interest. The motivation is very simple: first, as a rule the main use case for us is the synthesis of layouts for *multiple* structures with regard to a *single* fixed set of guidelines. Then, as we will see in a little while, providing a specialized version is simpler, than the generic one, and this version runs faster.

First we describe the construction of a non-relational, *functional* verifier. The guidelines description system matches the elements of the structure to a set of layout primitives. Thus, in order to verify that some layout respects the guidelines we have to *justify* each element of the layout by some guideline rules. More concretely, we identify two important notions:

- *Coverage*. We say, that layout *covers* a system if for any non-virtual element there is a layout primitive addressing this element. Coverage informally witnesses that no “real” element of the structure is left out unattended by the layout. The lack of coverage means that

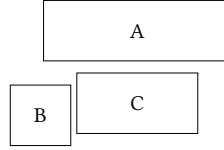


Figure 5. Layout Example: elements *A* and *B* are composed vertically, *A* and *C* are composed horizontally, *A* and *C* are aligned horizontally by center

there are some elements in the structure for which no placement constraints were derived according to the guidelines. We consider such layouts ill-formed.

- *Confirmation.* We say that a layout primitive occurrence p is *confirmed* by guidelines, if, first, there is an applicable rule in the guidelines which derives p and, second, all other layout primitives, derived by this rule, occur in the layout.

The coverage can be easily assessed by traversing all layout primitives, collecting all non-virtual elements and checking that set of all collected elements coincides with the set of all non-virtual elements in the structure.

The confirmation procedure traverses all layout primitives and tries to confirm each of them. This requires each rule of the guidelines to be *inverted*. For example, consider the following rule from JetBrains guidelines set:

$$X \xrightarrow{\text{descr}} Y, \text{ type } (Y, T) [\text{width } (Y) > K \wedge T \neq \text{checkbox}] \\ \Rightarrow \text{vert } (X, Y), \text{ halign } (X, Y);$$

It determines the following (sub)cases for the confirmation procedure $\text{confirm}_{\mathcal{G}}$:

$$\begin{aligned} \text{confirm}_{\mathcal{G}} (\mathcal{S}, \mathcal{L}) = \\ \text{vert } (X, Y) \in \mathcal{L} \rightarrow \\ \dots \\ X \xrightarrow{\text{descr}} Y \in \mathcal{S} \wedge \\ \text{type } (Y, T) \in \mathcal{S} \wedge \\ \text{width } (Y) > K \wedge \\ T \neq \text{checkbox} \wedge \\ \text{halign } (X, Y) \in \mathcal{L} \\ \dots \\ \text{halign } (X, Y) \in \mathcal{L} \rightarrow \\ \dots \\ X \xrightarrow{\text{descr}} Y \in \mathcal{S} \wedge \\ \text{type } (Y, T) \in \mathcal{S} \wedge \\ \text{width } (Y) > K \wedge \\ T \neq \text{checkbox} \wedge \\ \text{vert } (X, Y) \in \mathcal{L} \\ \dots \end{aligned}$$

Thus, building functional verifier is rather a simple routine procedure.

There are multiple ways to convert this verifier into relational form. One way is to use a relational interpreter for the functional language the verifier is implemented in [6]; another one is to apply *relational conversion* [19] which syntactically transforms functional programs into relational ones.

We tried the latter and, indeed, acquired a relational layout synthesizer with a reasonable behavior. There were, however, three problems:

1. Functional verifier used lists of layout primitives to represent layouts. Thus, there were multiple answers which, in fact, represented the same layouts due to permutations or repetitions of the elements of lists.
2. Incomplete answers: it turned out that often even a small number of layout primitives is enough to cover the structure and be confirmed by the guidelines. While according to our criteria this is not an invalid answer, it somewhat contradicts the informal semantics of the guidelines as it is expected that, on the contrary, as much rules as possible have to be used.
3. Performance: even for a simple problem the synthesis took tens of seconds to complete.

For the first problem we devised a specific representation for the layouts (“binary hypercube”) in which any layout can be represented uniquely. This representation in essence is a bitscale representation of sets and relations, and it has to be specifically defined for each structure. For example, let us have two UI elements *a* and *b* and, for simplicity, only two layout primitives *vert* and *hor*. Then the following definitions introduce the representation for layouts:

```
type rels    = struct {vert : bool; hor : bool}
type  $\alpha$  roles = struct {a :  $\alpha$ ; b :  $\alpha$ }
type layout = rels roles roles
```

This approach eliminates the extra overhead of the search for equivalent answers with different representations.

The problem of incomplete answers was solved using the following simple observation: as each guideline rule can only *add* some layout primitives, it is sufficient to filter out the answers which are *subsumed* by other answers. Thus, first we call the synthesizer for all answers (this is feasible as, due to the finiteness of the search space, the synthesis is refutationally complete), and then in a post-processing throw away all incomplete answers.

The last decision, however, makes the performance of the whole synthesis an issue. Fortunately, there turned out to

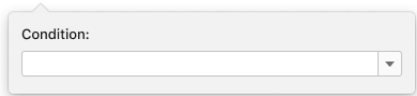

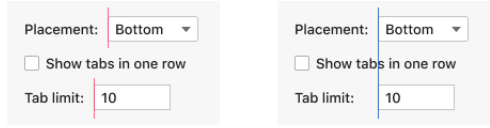
$X \xrightarrow{descr} Y, \text{ type } (Y, \text{checkbox})$ $\Rightarrow \text{hor } (Y, X);$ $X \xrightarrow{ord} Y$ $\Rightarrow \text{vert } (X, Y), \text{ halign } (X, Y);$ $X \xrightarrow{sub} Y$ $\Rightarrow \text{vert } (X, Y), \text{ indent } (X, Y);$	<p>There are no such explicit rules in the guideline, but these basic cases implicitly follow from the others.</p>
$X \xrightarrow{descr} Y, \text{ type } (Y, T) [\text{width } (Y) \leq K \wedge T \neq \text{checkbox}]$ $\Rightarrow \text{hor } (X, Y);$ $X \xrightarrow{descr} Y, \text{ type } (Y, T) [\text{width } (Y) > K \wedge T \neq \text{checkbox}]$ $\Rightarrow \text{vert } (X, Y), \text{ halign } (X, Y);$	<p>08 If an input box is long, and the horizontal space is limited, place the label above the box. Otherwise, always put the label and the box on the same line.</p>  <p>By the second rule variable X is a label with text “Condition” and variable Y is a drop-down list.</p>
$C \xrightarrow{ord} C',$ $C \xrightarrow{comp} Y,$ $C' \xrightarrow{comp} Y',$ $X \xrightarrow{descr} Y,$ $X' \xrightarrow{descr} Y' [\max (\text{width } (X), \text{width } (X')) < 2 * \min (\text{width } (X), \text{width } (X'))]$ $\Rightarrow \text{halign } (Y, Y');$	<p>05 By default, put input controls with labels of similar length on different lines and align their input boxes on the left side.</p>  <p>The rule applied twice. In the first case virtual control C contains the top textbox (Y) and virtual control C' contains the middle textbox (Y'). The textbox Y is described by the top label (X) and the textbox Y' is described by the middle label (X'). The second case is similar for middle and bottom elements.</p>
$C \xrightarrow{ord} C'',$ $C'' \xrightarrow{ord} C',$ $C \xrightarrow{comp} Y,$ $C' \xrightarrow{comp} Y',$ $X \xrightarrow{descr} Y,$ $X' \xrightarrow{descr} Y' [\max (\text{width } (X), \text{width } (X')) < 2 * \min (\text{width } (X), \text{width } (X'))]$ $\Rightarrow \text{halign } (Y, Y');$	<p>09 If there are two input controls with labels of similar length that are separated from each other by a single control, align their input boxes on the left side.</p>  <p>By the rule virtual control C contains the top textbox (Y) and virtual control C' contains the bottom textbox (Y'). There is a virtual control Y'' containing the checkbox between them. Labels X and X' describe textboxes Y and Y'.</p>

Figure 6. An Example of UI Guidelines Specification

be an easy and elegant solution. Note, the relational nature of structure description calls for its relational representation. Initially, in functional verifier, we used a list of relation specifications as a representation for structure. However, an arbitrary structure can explicitly be encoded as a specification in a relational programming language. For example, if we have the following structure

```

type (X, checkbox)
type (Y, label)
type (Z, textedit)
type (W, virtual)

Y  $\xrightarrow{descr}$  X
W  $\xrightarrow{comp}$  X
W  $\xrightarrow{comp}$  Y
W  $\xrightarrow{ord}$  Z

```

then we can directly convert it into the following set of relational definitions:

```

let typeo x y = ocanren {
  y == Checkbox & x == X |
  y == Label & x == Y |
  y == TextEdit & x == Z |
  y == Virtual & x == W
}

let descro x y = ocanren {
  x == Y & y == X
}

let compo x y = ocanren {
  x == W & {y == X | y == Y}
}

let ordo x y = ocanren {
  x == W & y == Z
}

```

After the conversion, a pattern to match over a structure can be turned into a conventional relational goal; for example, the pattern

$$X \xrightarrow{descr} Y, \text{ type } (Y, T) [\text{width } (Y) > K \wedge T \neq \text{checkbox}]$$

which we've already considered as example multiple times can be expressed by the following goal:

```

descro x y &
fresh t, w in
  typeo y t &
  widtho y w &

```

```

t < k &
t ≠ Checkbox

```

Not only this implementation is much faster than that obtained from a functional one via relational conversion, but is it much more native for the given problem.

To summarize, we implement the layout synthesizer in the following way:

- the synthesizer is specialized for a given set of guidelines and a given structure;
- the structure is converted into a set of relational definitions;
- a corresponding “binary hypercube” definition for layout representation is generated for the structure;
- the set of guideline rules is “inverted”, and all matching on the structure is turned into appropriate goals using the definitions generated for this structure.

This, coupled with the usual coverage check, delivers us the layout synthesizer. The drawback of this approach is that we need to re-generate the part of the system with each change of the structure; on the other hand the gains in the performance are essential. This is rather a natural trade-off when specialization techniques are used.

4 Solving Layout Constraints

Layout synthesizer for a given structure produces a set of layout primitives. This set in fact specifies a number of *integer linear constraints* which have to be solved in order to obtain the final, absolute coordinates of the elements. Indeed, let us introduce the notations for integer constants and *variables* which values have to be defined as a result of constraint solving:

- $C.\text{width}$ — the width of UI element C (integer constant for real UI elements and a *variable* for virtual UI elements);
- $C.\text{height}$ — the height of UI element C (integer constant for real UI elements and a *variable* for virtual UI elements);
- $C.x$ — the X -coordinate of UI element C (a *variable* for each UI element);
- $C.y$ — the Y -coordinate of UI element C (a *variable* for each UI element);
- a_H, a_V — the horizontal/vertical alignments between two UI elements (integer constant);
- i_H, i_V — the vertical/horizontal insets between two UI elements (integer constants);
- ind — indent between two UI elements (integer constant);
- W — the width of enclosing panel (integer constant);
- H — the height of enclosing panel (integer constant).

The set of layout primitives defines a set of integer constraints as follows.

- A layout constraint **hor** (C_1, C_2) introduces constraints

$$C_2.x - C_1.x \leq i_H + C_1.width$$

$$C_2.y - C_1.y = a_V$$

- A layout constraint **vert** (C_1, C_2) introduces constraints

$$C_2.x - C_1.x = a_H$$

$$C_2.y - C_1.y \leq i_V + C_1.height$$

- A layout constraint **halign** (C_1, C_2) introduces constraint

$$C_1.x - C_2.x = a_H$$

- A layout constraint **valign** (C_1, C_2) introduces constraint

$$C_1.y - C_2.y = a_V$$

- A layout constraint **indent** (C_1, C_2) introduces constraints

$$C_1.x - C_2.x = ind$$

$$C_1.y - C_2.y \leq i_V + C_1.height$$

Also, for each virtual UI element C containing UI elements C_1, \dots, C_n the following constraints are needed that define its actual coordinates, width and height:

$$C.x = \min \{C_1.x, \dots, C_n.x\}$$

$$C.y = \min \{C_1.y, \dots, C_n.y\}$$

$$C.x + C.width = \max \{C_i.x + C_i.width\}_{i=1}^n$$

$$C.y + C.height = \max \{C_i.y + C_i.height\}_{i=1}^n$$

Additionally, a number of inequality constraints is added which restrict the maximal possible values for coordinates taking into account the size of enclosing panel. For each UI element C we introduce the following constraints:

$$C.x \geq 0$$

$$C.y \geq 0$$

$$C.x + C.width \leq W$$

$$C.y + C.height \leq H$$

There exists a number of ways to solve the set of integer linear inequalities (for example, using SMT solvers over the linear integer arithmetic theory). It is, however, very appealing to try to employ relational verifiers yet again. This relational solver would have the following benefit: it could be seamlessly integrated into the layout synthesis procedure, thus allowing some constraint selection to be rejected at earlier stages.

However, the current relational implementation of this solver is underperforming in the presence of virtual UI elements. As mentioned above, the width and height of virtual

UI elements are not constants, which greatly increases the search space.

So, for the time being, we, indeed, used Z3 Theorem Prover [7] to determine the absolute coordinates of the UI elements (as well as the width and height of all virtual UI elements).

5 Implementation and Evaluation

In this section we describe the technical details of our prototype implementation and discuss several examples of layout synthesis for specific UI structures.

The prototype of our tool is developed as a client/server application using WEB technologies. On client side we use HTML5 for rendering and (mostly) OCAML [17] (compiled to JAVASCRIPT via JS_OF_OCAML [25]) for a client-server interaction. Server side is a native OCAML application equipped by OCANREN [16] and Z3. An initial attempt was to build a serverless prototype but, first, we observed that JS_OF_OCAML runs as twice as slower than the native OCAML for simple cases, and, second, running a C++ application Z3 in a WEB-browser is non-trivial.

5.1 Client Side

A web page is essentially a text area for the user input of structure information in the syntax similar to that in Fig. 7, and a space for rendering results. The client-server interaction runs in several phases. The client sends structure information to the server and receives possibly many layouts. After that it sends the layouts to the server one by one and receives the exact coordinates of the UI elements in the structure. The layouts which were successfully evaluated with Z3 are being rendered in the end.

5.2 Server Side

The server side is implemented in OCAML with assistance of OCANREN¹ [7], NOCANREN² [19] and Z3 [7]. It could be separated into two parts: synthesis of a layout and evaluation of absolute coordinates.

The synthesis of a layout is implemented mostly in functional style with assistance of NOCANREN, which translates a dialect of ML to OCANREN. The exception is a confirmation check (described in Section 3) which is easier to implement in relational style because it applies non-deterministically many guideline rules. Encoding of the guideline rules themselves in OCANREN is error-prone because of inversion requirement. We implemented an embedded DSL for OCAML for these guidelines which performs an inversion at compile time.

We could name two drawbacks in our current implementation. For now we have a limited amount of identifiers for UI elements (named by letters from A to J on Figure 7) which

¹<https://github.com/JetBrains-Research/OCanren>

²<https://github.com/Lozov-Petr/noCanren>

allows to represent our binary hypercube finitely, but reduces the expressivity of input structure definition. In future it shall be easily fixed because all possible structure definitions have a finite number of elements. Another drawback of our binary hypercube is an ability to encode nonsensical layouts (for example, one element subordinates another and vice versa). Right now these layouts are filtered out during the generation of absolute coordinates, but ideally we want them to be non-representable using type definitions of our synthesizer.

The second part of server implementation is the calculation of absolute coordinates with assistance from Z3. This stage filters out problematical layouts and performs much faster than our relational synthesis. But we linked OCAML and Z3 to a single executable anyway to avoid paying any costs for inter process communication.

5.3 Evaluation

As an evaluation of the implemented tool we performed a synthesis for several UI structures using the UI guideline described in the subsection 2.3. The evaluation results are presented in Fig. 7.

The test UI structures use relations between real UI elements and virtual ones described in Section 2.1. As one can see, a proper layout was synthesized for each test. Note, the synthesis time for each was no more than 5 seconds.

6 Related Works

In the field of relational programming one of the appealing applications is solving program synthesis problems. First of all, for the synthesis of programs with certain properties relational interpreters of programming languages are used. This approach allows one to fully or partially synthesize *quines* [6], or programs that pass a set of tests [5]. More generally, relational interpreters make it possible to synthesize solutions for search problems [18]. Also, relational theorem provers allow one to synthesize both proofs by theorems and theorems by proofs [2].

In the layout construction area there are some tools for declarative description of the layout [9, 14, 21] using domain-specific languages (DSL). These approaches, however, require explicit or implicit specification of the exact locations for the UI components. Therefore, these languages are not suitable for an abstract description of the interface. We are not aware of any approaches for synthesizing the layout according to the description completely abstracted from the information about the coordinates of the components.

7 Discussion and Future Work

We presented the preliminary results of our work on guideline-based synthesis of UI layouts. While this work is not yet finished, we consider our current results promising.

One interesting question which may arise is if the application of relational programming is essential for this problem to be solved. Indeed, the set of guidelines describes a matching (or rewriting) system which, in principle, can be directly implemented without any use of relational techniques. We argue, however, that in this case a whole piece of work on justification of the correctness and completeness of the solution would have been repeated anew. In our case, the justification trivially follows from the completeness of the MINIKANREN search and refutational completeness of our solution. We also speculate that such a solution would require reinventing of some implementation techniques to support nondeterminism and backtracking, which are already native to relational programming. Finally, the duality between patterns over structure and relational goals (initially unexpected for us), to our opinion, witnesses, that relational programming is a truly native technique for this problem, and we are not stretching an owl over a globe.

To describe the structure of the interface, we have developed our own DSL, since existing DSLs do not allow to completely abstract from the coordinates of elements. Our prototype for now supports rather a limited subset of structure/guideline description constructs. The most relevant task for the future is extending this subset to make it possible to express the real-world guidelines and synthesize layouts for real-world industrial UI components.

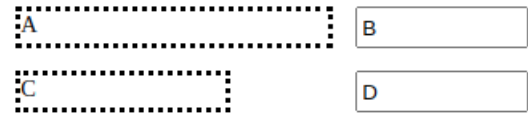
We can also consider the task of getting rid of extra solver (Z3) as relevant; this would not only simplify the infrastructure of the system, but would also allow to integrate the constraint resolution phase into the constraint synthesis, improving the performance of the whole system. To do this, we plan to develop a relational integer inequality solver with a binary representation of fixed-size numbers. Note that for satisfactory performance it will also be necessary to extend OCANREN with inequality operations for such numbers in the form of a new type of constraint.

Finally, the current DSL for structure description is difficult for the untrained user to use. Therefore, we consider the task of developing a more user-friendly DSL that will translate into the current one.

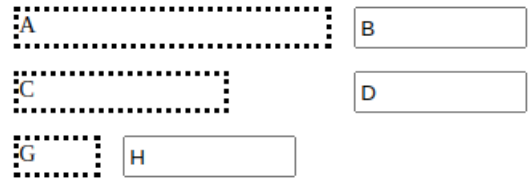
References

- [1] Smashing Magazine 2016. *The Thumb Zone: Designing For Mobile Users*. Smashing Magazine. Retrieved July 20, 2022 from <https://www.smashingmagazine.com/2016/09/the-thumb-zone-designing-for-mobile-users/>
- [2] Bernhard Beckert and Joachim Posegga. 1995. leanTAP: Lean Tableau-based Deduction. *Journal of Automated Reasoning* 15 (1995), 339–358.
- [3] Jacquelyn Bengfort. 2018. *Thin vs. Thick vs. Zero Client: What's the Right Fit for Your Business?* Technology Solutions That Drive Business. Retrieved July 20, 2022 from <https://biztechmagazine.com/article/2018/10/thin-vs-thick-vs-zero-client-whats-right-fit-your-business-perfcon>
- [4] Builder.io. 2011. *Figma to HTML, CSS, React & more!* Retrieved July 20, 2022 from <https://www.figma.com/community/plugin/747985167520967365/Figma-to-HTML%2C-CSS%2C-React-%26->

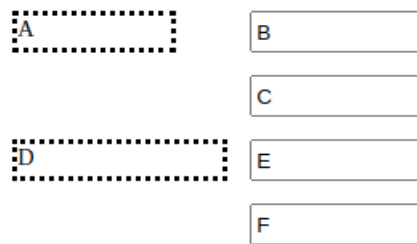
type (A, Label), type (B, TextEdit), type (C, Label),
 type (D, TextEdit), type (E, Virtual), type (F, Virtual),
 $A \xrightarrow{descr} B$, $C \xrightarrow{descr} D$, $E \xrightarrow{comp} A$, $E \xrightarrow{comp} B$, $F \xrightarrow{comp} C$, $F \xrightarrow{comp} D$,
 $E \xrightarrow{ord} F$ [width (A) = 96 \wedge width (B) = 64]



type (A, Label), type (B, TextEdit), type (C, Label),
 type (D, TextEdit), type (G, Label), type (H, TextEdit),
 type (E, Virtual), type (F, Virtual), type (I, Virtual),
 $A \xrightarrow{descr} B$, $C \xrightarrow{descr} D$, $G \xrightarrow{descr} H$, $E \xrightarrow{comp} A$, $E \xrightarrow{comp} B$,
 $F \xrightarrow{comp} C$, $F \xrightarrow{comp} D$, $I \xrightarrow{comp} G$, $I \xrightarrow{comp} H$, $E \xrightarrow{ord} F$,
 $F \xrightarrow{ord} I$ [width (A) = 96 \wedge width (B) = 64 \wedge width (G) = 16]



type (A, Label), type (B, TextEdit), type (C, TextEdit),
 type (D, Label), type (E, TextEdit), type (F, TextEdit),
 type (G, Virtual), type (H, Virtual),
 $A \xrightarrow{descr} B$, $D \xrightarrow{descr} E$, $B \xrightarrow{ord} C$, $E \xrightarrow{ord} F$, $G \xrightarrow{comp} A$,
 $G \xrightarrow{comp} B$, $G \xrightarrow{comp} C$, $H \xrightarrow{comp} D$, $H \xrightarrow{comp} E$, $H \xrightarrow{comp} F$,
 $G \xrightarrow{ord} H$ [width (A) = 48 \wedge width (B) = 64]



type (A, Label), type (B, TextEdit), type (C, Label),
 type (D, CheckBox), type (G, Label), type (H, TextEdit),
 type (E, Virtual), type (F, Virtual), type (I, Virtual),
 $A \xrightarrow{descr} B$, $C \xrightarrow{descr} D$, $G \xrightarrow{descr} H$, $E \xrightarrow{comp} A$, $E \xrightarrow{comp} B$,
 $F \xrightarrow{comp} C$, $F \xrightarrow{comp} D$, $I \xrightarrow{comp} G$, $I \xrightarrow{comp} H$, $E \xrightarrow{ord} F$,
 $F \xrightarrow{ord} I$ [width (A) = 64 \wedge width (B) = 64 \wedge width (G) = 48]

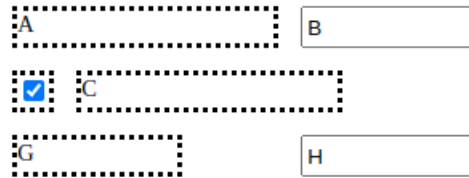


Figure 7. Examples of Structures and Synthesized Layouts

[more!](#)

- [5] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proceedings of ACM Program. Lang.* (2017), 8:1–8:26.
- [6] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) (Scheme '12). Association for Computing Machinery, New York, NY, USA, 8–29. <https://doi.org/10.1145/2661103.2661105>
- [7] Leonardo de Moura and Nikolaj Björner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
- [8] Daniel Deutsch. 2017. *Understanding MVC Architecture with React*. Retrieved July 20, 2022 from <https://medium.com/createdd-notes/understanding-mvc-architecture-with-react-6cd38e91fef4>
- [9] Android Developers. 2022. *Jetpack Compose*. Retrieved July 20, 2022 from <https://developer.android.com/jetpack/compose>
- [10] The Interaction Design Foundation. 2022. *Design Guidelines*. Retrieved July 20, 2022 from <https://www.interaction-design.org/literature/topics/design-guidelines>
- [11] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2005. *The Reasoned Schemer* (2nd ed.). The MIT Press. <https://doi.org/10.7551/mitpress/5801.001.0001>
- [12] Elizabeth Gerber and Maureen Carroll. 2012. The psychological experience of prototyping. *Design Studies* 33, 1 (2012), 64–84. <https://doi.org/10.1016/j.destud.2011.06.005>
- [13] Martin Haft, Bernhard Humm, and Johannes Siedersleben. 2005. The Architect’s Dilemma – Will Reference Architectures Help?. In *Quality of Software Architectures and Software Quality*, Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker, and Patrick J. Schroeder (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 106–122. https://doi.org/10.1007/11558569_9
- [14] Apple Inc. 2022. *SwiftUI*. Retrieved July 20, 2022 from <https://developer.apple.com/xcode/swiftui/>
- [15] ISO 9241-210:2019 2019. *Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems*. Standard. International Organization for Standardization.

- [16] Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. (2016), 1–22. <https://doi.org/10.4204/EPTCS.285.1>
- [17] Xavier Leroy, Damien Doligez, Jacques Garrigue Alain Frisch, Didier Rémy, and Jérôme Vouillon. 2020.
- [18] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational interpreters for search problems. In *Relational Programming Workshop*. 43.
- [19] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2017. Typed relational conversion. In *International Symposium on Trends in Functional Programming*. Springer, 39–58.
- [20] Trygve M. H. Reenskaug. 1979. *Thing-Model-View-Editor – an Example from a planningsystem*. Xerox PARC. Retrieved July 20, 2022 from <https://folk.universitetetioslo.no/trygver/1979/mvc-1/1979-05-MVC.pdf>
- [21] Inc Meta Platforms. 2022. *React: A JavaScript library for building user interfaces*. Retrieved July 20, 2022 from <https://reactjs.org/>
- [22] Jorge-Luis Pérez-Medina, Sophie Dupuy-Chessa, and Agnès Front. 2007. A Survey of Model Driven Engineering Tools for User Interface Design. In *Task Models and Diagrams for User Interface Design*, Marco Winckler, Hilary Johnson, and Philippe Palanque (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–97. https://doi.org/10.1007/978-3-540-77222-4_8
- [23] JetBrains s.r.o. 2000–2022. *IntelliJ Platform UI Guidelines*. Retrieved July 20, 2022 from <https://jetbrains.github.io/ui/>
- [24] JetBrains s.r.o. 2000–2022. *IntelliJ Platform UI Guidelines: Layout*. Retrieved July 20, 2022 from <https://jetbrains.github.io/ui/principles/layout>
- [25] Jérôme Vouillon and Vincent Balat. 2014. From Bytecode to JavaScript: the Js_of_ocaml Compiler. *Software: Practice and Experience* 44 (08 2014). <https://doi.org/10.1002/spe.2187>