Synthesis of Web Layouts from Examples

Dylan Lukes*
dlukes@eng.ucsd.edu
UC San Diego
La Jolla, CA, USA

John Sarracino*†
jsarracino@cornell.edu
Cornell University
Ithaca, NY, USA

Cora Coleman
ccoleman@eng.ucsd.edu
UC San Diego
La Jolla, CA, USA

Hila Peleg†
hilap@cs.technion.ac.il
Technion
Haifa, Israel

Sorin Lerner
lerner@eng.ucsd.edu
UC San Diego
La Jolla, CA, USA

Nadia Polikarpova
npolikarpova@eng.ucsd.edu
UC San Diego
La Jolla, CA, USA

ABSTRACT

We present a new technique for synthesizing dynamic, constraint-based visual layouts from examples. Our technique tackles two major challenges of layout synthesis. First, realistic layouts, especially on the web, often contain hundreds of elements, so the synthesizer needs to *scale* to layouts of this complexity. Second, in common usage scenarios, examples contain *noise*, so the synthesizer needs to be tolerant to imprecise inputs. To address these challenges we propose a two-phase approach to synthesis, where a *local inference* phase rapidly generates a set of likely candidate constraints that satisfy the given examples, and then a *global inference* phase selects a subset of the candidates that generalizes to unseen inputs. This separation of concerns helps our technique tackle the two challenges: the local phase employs *Bayesian inference* to handle noisy inputs, while the global phase leverages the *hierarchical* nature of complex layouts to decompose the global inference problem into inference of independent sub-layouts.

We implemented this technique in a tool called Mockdown and evaluated it on nine real-world web layouts, as well as a series of widespread layout components and an existing dataset of 644 Android applications. Our experiments show that Mockdown is able to synthesize a highly accurate layout for the majority of benchmarks from just three examples (two for Android layouts), and that it scales to layouts with over 600 elements, about 30x more than has been reported in prior work on layout synthesis.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; **Automatic programming**; **Programming by example**.

KEYWORDS

Program Synthesis, Constraint-based layouts, Web layouts

*Both authors contributed equally to this research.
†Work done while at UC San Diego.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468533

ACM Reference Format:
Dylan Lukes, John Sarracino, Cora Coleman, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. Synthesis of Web Layouts from Examples. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468533

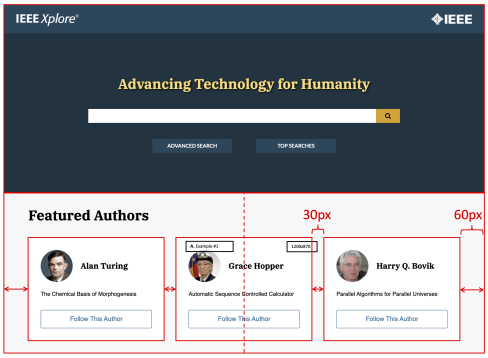
1 INTRODUCTION

Visual layout is the problem of arranging graphical user interface (GUI) elements on screen. Creating layouts is an integral part of developing desktop, mobile, and web-based applications. In all these domains, the layouts must be *dynamic*; that is, the absolute positions of the GUI elements, also called *views*, must adjust to the dimensions of the *root view*, in order to keep the application functional and aesthetically pleasing for a wide range of dimensions [40].

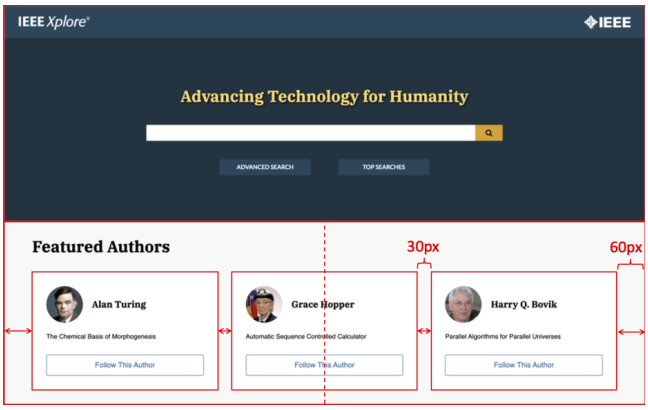
Constraint-Based Layout. For webpages, dynamic layouts are usually defined using Cascading Style Sheets (CSS) [39]. However, the expressivity of CSS is limited, for example in allowing sibling elements to be defined relative to each other. In mobile applications, this niche has been filled by *declarative constraint systems* such as Apple's AutoLayout [50] and Android's ConstraintLayout [27], powerful tools for implementing dynamic layouts. The key idea of declarative constraint systems is to express local relations between views, such as "adjacent-to" or "center-aligned", as *constraints*, and let the layout engine solve these constraints at run time to figure out where to place each GUI element given particular root view dimensions. For example, Fig. 1a shows a screenshot of a slightly simplified version of the IEEE Xplore website [1], which includes a "Featured Authors" panel displaying profiles of three authors. The desired horizontal layout for this panel can be expressed with constraints stating that the middle profile is centered in the page and the margins on either side of each profile are fixed. At run time, as the user resizes the page, the layout responds according to the constraints (see Fig. 1b): the width of the profiles changes, preserving centering and margins.

While constraint-based layouts have been greatly successful in the mobile domain, they are not as prevalent for web design. A potential reason is that a webpage has many more moving parts than a mobile application, and the relations between elements are far more complex, making constraints harder to author.

Layout Synthesis. A promising approach to easing the creation of flexible constraint-based layouts is to *synthesize* them automatically from examples. For instance, a layout synthesizer might take as



(a) Layout example 1: 1200x870



(b) Layout example 2: 1600x870

Figure 1: Snapshots of the IEEE Xplore website layout at different page dimensions.

input the two static "snapshots" in Fig. 1, and infer a system of constraints that (1) *matches the examples:* places all view just like in the input snapshots given their corresponding dimensions, and (2) *generalizes beyond the examples:* produces some "reasonable" placement for other dimensions. Layout synthesis enables various new ways of authoring dynamic layouts: for example, a designer may create several mock-ups in a direct-manipulation editor and then automatically convert them into a general constraint-based layout; alternatively, a user can emulate the look of an existing website, by *scraping* layout snapshots directly from the browser's rendering of the HTML.

Challenges. In the mobile domain, the state of the art in layout synthesis is InferUI [11, 36], a tool for generating AutoLayout constraints for Android applications. At a high level, InferUI works by encoding the problem as an SMT query following the general approach of symbolic program synthesis [33, 54]. Extending layout synthesis beyond mobile apps, most notably to the web domain, poses two major **challenges**: (1) *Scalability:* web layouts typically have at least an order of magnitude more views than Android layouts do; the pure SMT-based approach scales poorly with the number of views in the layout. (2) *Noise:* in a realistic setting, both scraped and user-designed examples might contain noise: that is, the input snapshots satisfy the desired constraints *only approximately*; the pure SMT-based approach cannot handle such noisy inputs.

Our Solution. In this work we develop Mockdown, a new layout synthesizer that accepts layout examples (e.g., from a direct manipulation editor or a scraping tool) and finds the constraints needed. These can then be passed on to a web constraint-based layout engine (i.e., the web equivalent of AutoLayout equivalent) or be solved in JavaScript code to set the sizes and positions of views upon resize. Mockdown tackles the two challenges outlined above and is able to synthesize complex layouts from realistic noisy inputs. In contrast to the the monolithic SMT encoding of prior work, the synthesis process in Mockdown is split into two phases: *local inference*, which considers each constraint in the search space in isolation, and *global inference*, which considers systems of constraints as a whole. The goal of local inference is to suggest a set of *candidate constraints*, which match the provided examples, but do not necessarily generalize to unseen dimensions; the goal of global inference is to pick a subset of the candidates that generalizes. For example, given just the one example in Fig. 1a, Mockdown's local inference phase would suggest that all author profiles have a fixed width of 340 pixels *and* that their outer and inner margins are fixed (at 60 and 30 pixels, respectively). While these constraints are true for this example, they cannot all continue to hold if we resize the page. Mockdown's global inference uses a MaxSMT [12] solver to pick a maximal subset of these constraints that is *satisfiable* for a wide range of page dimensions; in this case, the global inference must get rid of either the fixed width constraints or one of the margin constraints.

Handling Noise via Bayesian Local Inference. Mockdown's two-phase architecture helps it address the two major challenges outlined above. To deal with noise, the local phase employs *Bayesian inference* to generate constraint candidates that, on the one hand, are likely to generate the data, and on the other hand, are simple according to a domain-specific prior. Crucially, the SMT-based global phase never gets exposed to noisy data.

Scaling via Hierarchical Global Inference. With Mockdown's two-phase approach, the local phase rapidly falsifies unlikely constraints, thereby pruning the search space for the computationally intensive global phase. Our experiments show, however, that this pruning is not sufficient to make the global phase scale to complex web layouts. To further improve performance, we leverage the observation that real-world layouts are *hierarchical*: the views are arranged in a tree, and the layout of the upper layers of the hierarchy is independent from the lower layers. For example, for the IEEE website, we first infer the layout of the three author profiles and then independently infer the layout of their contents (image, name, "Follow..." button). This *hierarchical global inference* significantly improves scalability for real-world layouts, where the number of "sibling" views is typically small.

Contributions. In summary, this paper makes the following contributions:

- A *two-phase layout synthesis algorithm*, where a local phase rapidly generates the set of likely candidate constraints that satisfy the examples, and a global phase selects a generalizable subset of these constraints (Sec. 2).
- *Noise-tolerant local inference* via Bayesian learning (Sec. 3).

- *Hierarchical global inference*, which leverages the structure of real-world layouts to achieve scalability (Sec. 4).
- An implementation of the two-phase synthesis algorithm in a tool called Mockdown, which has been evaluated on real-world Android and web layouts (Sec. 5). Mockdown can discover the constraints of realistic layouts with as much as an order of magnitude more elements as previous tools.

2 MOTIVATING EXAMPLE

Consider the task of designing a dynamic layout for the IEEE Xplore website introduced in Fig. 1. In this section we explain how Mockdown can infer a constraint-based layout for this website using two examples (with the focus on laying out the author profiles within the "Featured Authors" pane). The inference process is depicted in Fig. 2, to which we refer throughout the section.

2.1 The Mockdown Interface

Mockdown takes as input a set of *examples*, *i.e.* static snapshots of the layout at fixed page dimensions. Concretely, every input example is a JSON file that encodes the hierarchy of views and their absolute positions within the page. For example, Fig. 2A shows an excerpt from the Mockdown inputs for the two snapshots from Fig. 1, focused on the "Featured Authors" pane. Across different examples, the set of views and their hierarchy is assumed to be the same (here, the author profile views `turing` and `hopper` are children of the `authors` pane), while their positions, specified via the `rect` attribute, differ.

Given these two examples, Mockdown infers a set of constraints shown in Fig. 2F. These constraints represent linear relationships between *anchors* of the views, where an anchor is either an edge, like `hopper.left`, or a dimension derived from the edges, like `hopper.width` or `hopper.center_x`. Constraints of this form serve as input to a *constraint layout engine*, which, given a new desired page size (for example 1300×870), will generate a concrete placement of all the views that satisfies the constraints, as shown in Fig. 2G and Fig. 2H.

Inputs to Mockdown might come from a variety of front-ends. Perhaps most obviously, a designer might use a direct-manipulation tool to manually place the views and specify their hierarchy. Alternatively, if a user would like to replicate the look of an existing website, they can *scrape* this information from this website's HTML rendered in the browser; in fact, for evaluation purposes we have implemented such a scraping front-end for Mockdown. With either front-end, we can envision an interactive design process where the user starts with a single example and then iteratively adds more examples if upon visual inspection at a new page dimension they are dissatisfied with the generated dynamic layout.

Likewise, the outputs of Mockdown might be consumed by a variety of layout engines, including the popular engines AutoLayout [50] and ConstraintLayout [27] for the mobile domain, and Grid Style Sheets [58] for the web domain. Because the support for constraint-based layout in web browsers is not (yet) widespread, for evaluation purposes we have implemented a simple layout engine that can be embedded into the webpage as JavaScript, and is based on the Kiwi [49] constraint solver[1].

[1] https://github.com/MangoTeam/mockdown-client

In the remainder of this paper we focus on the synthesis backend—*i.e.* going from the JSON examples to constraints—which is common for all front-ends and constrained layout engines. In the rest of this section, we detail the two phases of the constraint inference process: *local* and *global* inference.

2.2 Local Inference

The goal of local inference is to generate a set of *candidate constraints* matching the provided examples. An excerpt from the output of local inference is shown in Fig. 2D. This task can be solved efficiently following the invariant inference approach pioneered by Daikon [22]. This approach first instantiates constraint templates using variables in scope to produce *constraint sketches*, and then fills in the parameters of each sketch (or falsifies it altogether) based on the variable values observed in the examples. Mockdown's local inference is inspired by this approach, but it uses novel domain-specific techniques both for template instantiation and for parameter inference, which enable Mockdown to prune the space of candidate constraints and handle noise.

Visibility-Guided Template Instantiation. Although our algorithm is able to infer a variety of constraints—including offsets, alignments, size-ratios, and constant sizes—all of these constraints are in fact instances of single template $\mathtt{y} = a \cdot \mathtt{x} + b$, where `x`, `y` are anchors and a, b are rational constants. Mockdown instantiates this template with concrete anchors from the input examples to generate a set of constraint sketches, i.e. constraints where one or both of a and b are left as parameters, as shown in Fig. 2C. Similarly to type-based template instantiation in Daikon, Mockdown only generates *well-formed* sketches, following domain-specific rules that reflect constraint forms used in real-world layouts. For example, we generate an "offset" sketch `hopper.left` $= a \cdot$ `turing.right` $+ b$, but we do not generate sketches `hopper.left` $= a \cdot$ `turing.right` or `hopper.left` $=$ `turing.width` $+ b$, because edge ratios and edge-width relations are considered ill-formed.

Even after eliminating ill-formed sketches, the number of sketches generated for realistic inputs can be very large. To further prune the space of sketches we observe that real-world layouts place views with respect to their immediate parents and neighbors. For example, in Fig. 2 we do not want to relate `turing.right` to `bovik.left` because these two views are not immediate neighbors. To formalize this intuition, Mockdown builds a *visibility graph* at each level of the view hierarchy, and uses it to generate constraints only between visible edges of parent-child or sibling-sibling view pairs (two edges are considered visible if it is possible to draw a straight uninterrupted line between them). The visibility graph for our example is shown in Fig. 2B. Visibility-based pruning reduces the number of sketches generated for the IEEE example from 8820 to 576.

Bayesian Parameter Inference. For each generated sketch, Mockdown next learns its parameters a, b, such that the resulting constraint is satisfied for each input example. The major challenge for parameter inference is *noise* in the input data. For example, as shown in Fig. 2A, the anchor `turing.left` has values 59.5px and 60.15px in the two examples. The intended margin for this view is actually 60px; however, most use cases for layout synthesis inevitably introduce some amount of noise in the examples, either due to user input or due to numerical error during rendering and

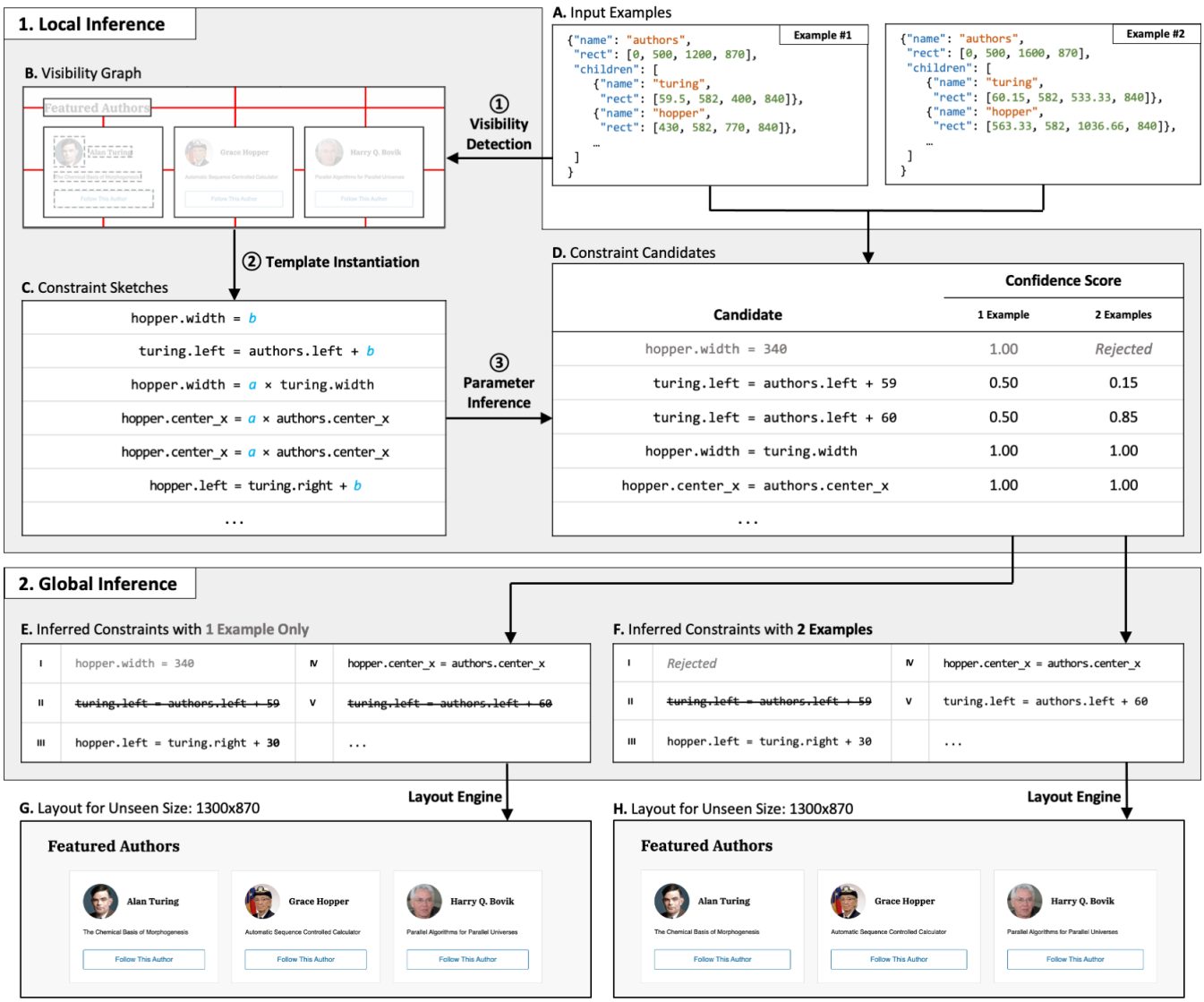


Figure 2: An overview of the constraint inference process in Mockdown.

scraping (recall that all numbers in JavaScript are represented as floats). A traditional, Daikon-like approach would instantiate the sketch `turing.left` = `authors.left` + b with $b = 59.5$ given the first example, and then would *falsify* this sketch given both examples, erroneously rejecting this desirable offset constraint.

Instead, Mockdown's *Bayesian parameter inference* algorithm turns each sketch into a set of candidate constraints that match the data *approximately*. To this end, Mockdown performs constrained linear regression, and then suggests candidate parameters, drawn from a domain-specific prior distribution, that fall within a confidence interval about the linear fit. In our example, even though the value $b = 59.5$ fits the data perfectly in the one-example case, our prior on offsets constrains them to be integers, and hence parameter inference produces two candidate values, $b = 59$ and $b = 60$ (see Fig. 2D). Mockdown also produces a *confidence score* for each candidate based on a Bayesian likelihood model; in this case both candidates have the score of 0.5, since our offset prior has no preference among different integers and both offsets fit the data equally well. After adding the second example, however, the value $b = 60$ receives a higher score, because now it fits the data much better.

2.3 Global Inference

Layout synthesis must produce constraints that *generalize* to unseen page sizes (within a certain range). In our example, the user expects the constraints produced by Mockdown to have a solution—i.e. to generate a placement for all views on the page—when the page width ranges between 1280px and 1680px. Candidate constraints produced by local inference (Fig. 2D) do not necessarily satisfy this criterion for two reasons. First, a subset of constraints might be (approximately) satisfied by all provided examples, but

be *unsatisfiable* on unseen sizes. This happens most often when constraints are inferred from one example: in Fig. 2D, given one example, we infer both the fixed width constraints for profiles, such as `hopper.width` = 340 and the fixed margin constraints between them; these constraints together are unsatisfiable for any width except 1200px. Second, due to noise, Mockdown sometimes infers incompatible constraint candidates from the same sketch: for example, the two offset constraints for `turing.left` with different values of b together are unsatisfiable.

The goal of global inference to select an optimal subset of the candidate constraints that is satisfiable on a range of sizes of interest. We consider a subset optimal if it maximizes the *total confidence score* of the constraint set. Intuitively, we prefer larger constraint sets to minimize the ambiguity in view placement, and among incompatible constraints we prefer those with higher confidence score, since they are more likely to have generated the data. To find the optimal subset, Mockdown uses a MaxSMT solver [12], to select the maximally satisfiable set of constraints on a random sample of input dimensions from a user-specified range. Fig. 2E shows that querying the MaxSMT solver with the candidates constraints inferred from one example on five unseen sizes (including 1300x870) leads to eliminating constraints II and V, which fix the outer margin. This produces a layout that generalizes to a range of sizes, albeit not in the way the user intended. With two examples, as shown in Fig. 2F, the fixed width constraint I has been rejected by the local phase, hence the only unsatisfiable subset of candidates is the two fixed-margin constraints II and V; global inference picks constraint V since it has a higher confidence score.

Hierarchical Decomposition. For layouts with many views, the MaxSMT queries can be computationally intractable: global inference as described so far takes over 2 min for the simplified IEEE layout and times out after 10 min on most real-world websites in our experiments. To address this challenge, we once again take advantage of layout-specific intuition: incompatible constraints typically appear *within the same level* of the view hierarchy. Based on this intuition, Mockdown's *hierarchical global inference* performs constraint selection independently for each level of the hierarchy. For example, in Fig. 2E all the conflicting constraints are between `authors` and its immediate children (`turing`, `hopper`, and `bovik`), hence there is no need to also consider the constraints between `turing` and its children (photo, name, etc) in the same MaxSMT query. In this example hierarchical inference reduces the solving time from 130 secs to 90 secs.

3 LOCAL INFERENCE

The local inference phase of Mockdown takes as input a view hierarchy and a set of input examples, and produces a set of *candidate constraints*. As we explained informally in Sec. 2.2, this phase consists of three steps: (1) detecting visibility between views; (2) generating a set of *constraint sketches* based on the view hierarchy and visibility information; and (3) filling the sketches with concrete numeric parameters based on input examples. Since the first two steps are relatively straightforward, in this section we focus on the third step, noise-tolerant parameter inference.

Parameter inference starts from a set of sketches $\mathcal{S} = \{s_1, \ldots, s_N\}$ and a set of examples $\mathcal{E} = \{e_1, \ldots, e_M\}$. Here each sketch s is of the form $\mathtt{y} = A \cdot \mathtt{x} + B$, where $\mathtt{x}, \mathtt{y} \in \mathcal{X}$ are *anchors* and $A, B \in \mathcal{P} \cup \{0\}$ are unknown *parameters* to be instantiated (or 0 if the parameter does not make sense for this pair of anchors); each example $e \in \mathcal{X} \to \mathbb{Q}$ maps anchors to a rational values. The output of parameter inference is a set of candidate constraints $C = \{c_1, \ldots, c_k\}$, where each c is of the form $\mathtt{y} = a \cdot \mathtt{x} + b$ ($a, b \in \mathbb{Q}$), such that c both (1) instantiates a sketch $\mathtt{y} = A \cdot \mathtt{x} + B$ from $\mathcal{S}$, and (2) approximately matches the examples $\mathcal{E}$, *i.e.* $\forall e \in \mathcal{E}. e[\mathtt{y}] \approx a \cdot e[\mathtt{x}] + b$. We formalize the meaning of this approximate match below using a Bayesian model.

Example. As a running example for the rest of this section, consider a parameter inference problem with a single sketch `child.width` = $A \cdot$ `parent.width`[2] and two examples $\mathcal{E} = \{e_1, e_2\}$, mapping the values of `(parent.width, child.width)` to `(100, 33)` and `(200, 67)`, respectively. Assume that the *true* intended value of A is 1/3; you can see that the examples are *noisy* on account of being rounded to the nearest whole pixel.

Naive Parameter Inference. The simplest approach to parameter inference is Daikon-style invariant detection [22]. That is, for each sketch, initialize a candidate constraint from the first example, and then iterate through the remaining examples, discarding the candidate if a new example does not match. This simple approach, however, fails in the presence of noise: in our running example, it would instantiate $a = 0.33$ using e_1, and then *reject* this constraint upon inspecting e_2, because $0.33 \cdot 200 = 66 \neq 67$ (this problem arises even with precise rational arithmetic, and is exacerbated by the more common floating-point arithmetic). To tackle this issue we develop a noise-tolerant parameter inference algorithm.

Bayesian Model. In the presence of noise, we cannot require the constraints C to match the data $\mathcal{E}$ exactly. Instead we need to find parameter values that *best explain* the data. The simplest way to formalize this intuition is as a *linear regression* problem, *i.e.* minimizing the *mean square error* between the line $\mathtt{y} = a \cdot \mathtt{x} + b$ and the examples in $\mathcal{E}$. Although linear regression can be performed efficiently using standard techniques, it rarely yields desirable results in our domain, because it returns arbitrary real (in practice, floating point) numbers for a and b, whereas we want to bias the inference towards simple rational parameters. Consider our running example: with a slight perturbation of input values, linear regression might return $a = 0.32323$, which is very unlikely to be what the designer intended, because the constant is too "complex". Even if we round this number to the closest rational with a denominator $D \leq 100$, we would still get a "complex" result $a = 32/99$ instead of a much simpler, and therefore preferable, result $a = 1/3$, which fits the data only slightly worse. We can formalize this trade-off between simplicity and fit using a Bayesian model.

In this formulation, our goal is to maximize the posterior probability $P(c \mid \mathcal{E})$ of a constraint given data, which by the Bayes' law is proportional to the prior probability $P(c)$ of the constraint and its fit to data $P(\mathcal{E} \mid c)$:

$$P(c \mid \mathcal{E}) \propto P(c) \cdot P(\mathcal{E} \mid c) \tag{1}$$

This problem is known as *Bayesian linear regression*. Here fit to data is easy to compute using the mean square error between the constraint and the data: $P(\mathcal{E} \mid c) \propto \exp(-\mathtt{mse}(c, \mathcal{E}))$; the challenge

[2]Note that $B = 0$ is fixed in this sketch, because Mockdown's sketch generation phase only considers ratio constraints between widths.

is to define the prior $P(c)$ such that it gives preference to simpler parameters, *e.g.* prefers 1/3 over 32/99 in our running example.

Simplicity Prior. We formalize our intuitive notion of complexity for a rational number q as its *Stern-Brocot depth*, or $\texttt{sb_depth}(q)$. The `sb_depth` of q is its depth in the *Stern-Brocot tree* [13, 55], which can be seen as a grammar for representing all rational numbers as sequences of left/right steps: *e.g.* $1/2 = L$, $1/3 = LL$, $2/3 = LR$, and so on. Hence, `sb_depth` measures the size of a "program representation" of a rational and approximates its complexity, similarly to traditional size-based complexity metrics in program synthesis. In particular, in our example $\texttt{sb_depth}(1/3) = 2$ while $\texttt{sb_depth}(32/99) = 16$, and hence the latter is considered much more complex.

Conveniently $\texttt{sb_depth}(q)$ is easy to compute: it is equal to the sum of the continued fraction representation of q [26]. We can thereby avoid computing or traversing the Stern-Brocot tree, and instead merely precompute the set of all rationals with denominator less than some D (our implementation uses $D = 100$). This sequence is known as a *Farey sequence* of order D.

We define the overall prior $P(\texttt{y} = a \cdot \texttt{x} + b) = P_m(a) \cdot P_a(b)$, where the prior P_m for multiplicative parameters is defined as a minimum description length prior [48], using `sb_depth` as the description length. The prior P_a for additive parameters is simpler: it is uniform over all integer values and zero on non-integer values.

Noise-Tolerant Parameter Inference. Given our complex prior, the Bayesian linear regression problem (1) cannot be solved analytically. We propose an algorithm to solve this problem efficiently but approximately, based on two domain-specific insights. First, there is a unique model c^* (with parameters (a^*, b^*)) that maximizes the data fit $P(\mathcal{E} \mid c^*)$, which can be found efficiently using traditional linear regression; since the posterior is proportional to the data fit, we can search for its maximum in the vicinity of c^*. Second, given our prior $P(c)$, there are only finitely many parameters (a, b) with a non-zero probability in any bounded box (because a must be a rational with maximum denominator $D = 100$ and b must be an integer). Based on these observations, our algorithm infers the (approximately) optimal model c by first computing c^* based only on the data fit, and then enumerating and scoring all pairs (a, b) with non-zero prior that are sufficiently close to (a^*, b^*).

Our algorithm is depicted in Fig. 3. Line 2 performs least-squares linear regression, which is constrained by the sketch (*e.g.* $B = 0$ in our running example), but ignores the prior. Regression yields three outputs: the model $c*$ (*i.e.* a pair of parameters (a^*, b^*)), the mean square error `mse` between $\mathcal{E}$ and $c*$, and a confidence interval `conf_int`. The `mse` is used to determine whether or not to reject the sketch (on lines 3-4): if it is large, indicating that our examples do not appear to fit a line, we reject the sketch. In our example, `mse` = 0.3 and the sketch is accepted; however, with the addition of a third data point that does not continue the line, *e.g.* `parent.width = 300` and `child.width = 67`, `mse` would rise to 105 and the sketch would therefore be rejected.

Lines 6–9 iterate over the relevant region of candidate parameters, score them according to the posterior, and store them in `scored_params`. We compute the region to iterate over as the intersection of `conf_int` and `param_space`, which represents parameters with non-zero prior; as we argued above, this intersection is always finite. In our running example, we obtain a `conf_int` of $[0.3215, 0.3385]$,

```
1: procedure InferParameters(s, ℰ)
2:     c*, mse, conf_int ← constrained_linreg(s, ℰ)
3:     if mse > cutoff then
4:         return []
5:     scored_params ← [ ]
6:     for c ∈ (conf_int ∩ param_space(s)) do
7:         score ← P(c) · P(ℰ | c)
8:         Add (c, score) to scored_params.
9:     return scored_params
```

Figure 3: Noise-Tolerant Parameter Inference algorithm.

which yields a set of candidate values for the parameter a, including: 32/99, 33/100, and the desired 1/3.

4 GLOBAL INFERENCE

The goal of Mockdown's global phase is to pick a subset of candidate constraints that generalizes to unseen dimensions. More formally, this phase takes as input a set of candidate constraints C and a score map $\texttt{score} \in C \to (0, 1]$ (both produced by the local inference), as well as a set $\texttt{dims} = \{(w_1, h_1), \ldots, (w_K, h_K)\}$ of *generalization dimensions*, where each dimension $(w_j, h_j) \in \mathbb{Q}^2$ is a pair of width and height of the root view. It computes a *maximal satisfiable subset* of C, *i.e.* a set $C' \subseteq C$ such that:

(1) C' is *satisfiable* on all dimensions `dims`, *i.e.* $\forall (w_j, h_j) \in \texttt{dims}$ there exists a substitution $\sigma_j \in \mathcal{X} \to \mathbb{Q}$ for all anchors in C', such that $\sigma_j[\texttt{root.width}] = w_j$, $\sigma_j[\texttt{root.height}] = h_j$ and $\forall c \in C'.\sigma_j(c)$ holds.
(2) any satisfiable subset C'' has at most the same score as C': $\sum\{\texttt{score}[c] \mid c \in C''\} \leq \sum\{\texttt{score}[c] \mid c \in C'\}$.

4.1 Selecting Constraints with MaxSMT

To select the maximal satisfiable subset of C, Mockdown relies on a *MaxSMT* solver [12]. For our purposes, a MaxSMT *query* consists of quantifier-free formula ϕ, a set of boolean *control variables* b_i, and a map `weight` from b_i to numeric weights. A MaxSAT solver returns `UNSAT` if ϕ is unsatisfiable, and otherwise returns a model $\mathcal{M}$ of ϕ which maximizes $\sum\{\texttt{weight}(b_i) \mid \mathcal{M}[b_i] = \texttt{true}\}$, *i.e.* the total weight of control variables set to true[3].

We encode the problem of finding C' as a MaxSMT query. To this end, we introduce the following variables:

(1) for each constraint $c_i \in C$, we introduce a boolean control variable b_i with $\texttt{weight}[b_i] = \texttt{score}[c_i]$; b_i represents whether c_i should be included in C';
(2) for each pair of anchor $x \in \mathcal{X}$ and dimension $(w_j, h_j) \in \texttt{dims}$, we introduce a rational variable x_j to represent the placement of anchor x under this dimension.

Next we construct the formula ϕ as a conjunction of the following sub-formulas:

(1) for each dimension j: $\texttt{root.width}_j = w_j \wedge \texttt{root.height}_j = h_j$;
(2) for each constraint i and dimension j: $b_i \Rightarrow c_i^j$, where c_i^j is the constraint c_i with each anchor x replaced by x_j;

[3] Alternatively, a MaxSMT solver can also be used to optimize a given objective function subject to a logical formula; we use this functionality in the next section.

```
1:  procedure SynthHier(C, dims, score)
2:      worklist ← [(root, dims)]
3:      C′ ← {}
4:      while worklist ≠ [] do
5:          (focus, dims_f) ← worklist.pop
6:          C_f ← restrict(focus, C)
7:          C′_f ← select(C_f, focus, dims_f, score)
8:          C′ ← C′ ∪ C′_f
9:          for child ∈ children(focus) do
10:             dims_c ← calc_dims(focus, dims_f, child, C′)
11:             worklist.push((child, dims_c))
12:     return C′
```

Figure 4: Mockdown Global generalization algorithm

(3) finally, we add domain-specific facts about anchors, such as $x_j >= 0$ and $x.\texttt{width}_j = x.\texttt{right}_j - x.\texttt{left}_j$.

Given this query, the solver attempts to find a placement for all anchors under each dimension in `dims` that satisfies constraints in C; if it cannot satisfy all the constraints, it is free to set some b_i to false, rendering all formulas of the form $b_i \Rightarrow c_i^j$ vacuous, and effectively disabling the constraint c_i. When the solver returns a model $\mathcal{M}$, we compute the result C' as the set of all constraints whose corresponding control variables are set to true: $C' = \{c_i \mid \mathcal{M}[b_i] = \texttt{true}\}$. This set is satisfiable because by construction of ϕ we know that $\forall c_i \in C'$, $\mathcal{M}(c_i^j)$ holds, and hence we can extract the substitution σ_j from the model $\mathcal{M}$ by taking $\sigma_j[x] = \mathcal{M}[x_j]$ for every anchor x. This set is also maximal because the solver guarantees to maximize the total weight of control variables set to true, which is equal to the total score of chosen constraints.

4.2 Hierarchical Decomposition

While the MaxSMT encoding of Sec. 4.1 is simple and works well for small layouts, it is intractable on realistic web layouts, since the query ϕ becomes too large to solve efficiently. To improve performance, we leverage a common layout design principle in which the layout of a view is determined only by the layout of its immediate parent and siblings. Based on this principle, we can decompose the global MaxSMT query into smaller queries for individual levels of the view hierarchy. More specifically, instead of solving the global query ϕ, which specifies the layout of all anchors $\mathcal{X}$, our algorithm iteratively solves sub-queries ϕ_x for each anchor $x \in \mathcal{X}$, where ϕ_x only specifies the layout of the direct children of x, and so is generally much smaller (and thus significantly easier to solve) than the global ϕ. We present this algorithm in Fig. 4.

The algorithm maintains a worklist of sub-problems, where each subproblem is represented as a pair $(\texttt{focus}, \texttt{dims}_f)$ of a *focus view* and its generalization dimensions. The worklist is initialized with the root view `root` and input dimensions `dims`. In each iteration, the algorithm picks a sub-problem $(\texttt{focus}, \texttt{dims}_f)$ from the worklist, and computes the set C'_f, which is the maximal subset of layout constraints for the *direct children* of `focus` satisfiable across $\texttt{dims}_f$. To this end, $\texttt{restrict}(\texttt{focus}, C)$ picks out the set $C_f \subseteq C$ of candidate constraints that mention the anchors of the direct children of `focus`, and $\texttt{select}(C_f, \texttt{focus}, \texttt{dims}_f, \texttt{score})$ invokes the MaxSMT constraint selection procedure from Sec. 4.1, restricted to the candidates C_f and using `focus` as the root view. The algorithm then adds the selected constraints C'_f to the output. Finally, lines 9–11 add a new sub-problem to the worklist for each direct child of `focus`.

In this last step, the challenge is to pick the right generalization dimensions $\texttt{dims}_c$ for the sub-problem rooted at `child`. If the range of dimensions is too narrow, the algorithm might return a set C' that is not actually satisfiable; if the range is too broad, the algorithm might return a set that is not maximal. To address this challenge, we again use a MaxSMT solver to calculate the lower and upper bound on the dimensions of `child`, given the currently chosen layout constraints C'. For example, to calculate the upper bound, the algorithm creates a MaxSMT query with the maximization objective $(\texttt{child.width}, \texttt{child.height})$, subject to C' and the upper bound on the dimensions of the parent `focus`. The dimensions $\texttt{dims}_c$ are then chosen at equal intervals between the lower and the upper bound. We encode this query in the function `calc_dims` in line 10.

5 EVALUATION

Mockdown is implemented in Python [41, 45, 51, 61] and TypeScript. We use Z3 version 4.8.7 [12, 17] as a MaxSMT solver and a wrapper[4] around the Kiwi [49] fork of Cassowary (in JavaScript) as the constraint layout engine. Our implementation[5] and our benchmarks and analysis scripts[6] are open and publicly available.

Research Questions. In our evaluation, we seek answers for the following research questions.

(RQ1) Does Mockdown enable layout synthesis for a broad variety of web applications?
(RQ2) Is noise-tolerant inference necessary for layout inference?
(RQ3) Does hierarchical decomposition help Mockdown to scale with layout size?
(RQ4) How general is Mockdown, *i.e.* is it applicable to domains beyond the web?

5.1 Benchmarks and Metrics

Benchmarks. We evaluate Mockdown on a combination of benchmarks from previous tools in the literature and our own curated benchmark suite. One challenge in evaluating a tool like Mockdown is assessing its behavior on websites of realistic scale and structure. We collect our realistic benchmarks from existing webpages using a web scraper implemented in JavaScript[7], which returns a JSON in the format shown in Fig. 2. We likewise translate benchmarks from previous tools into this JSON format. For RQ1, RQ2, and RQ3, we evaluate Mockdown on (1) a set of real, full webpages (*macro*benchmarks), and (2) *micro*benchmarks, small benchmarks that test specific aspects of layout synthesis. For RQ4, we additionally use a suite of Android layouts from the literature [36].

Macrobenchmarks. We collected a set of *nine* real-world websites, listed in Tab. 1, using our scraper. For each webpage, we identified a range of page sizes in which the layout behaves linearly, and scraped examples within this range. The dimensions for both train and test examples were selected randomly within the range. Our benchmark websites cover a variety of applications: `author` is a personal page

[4] https://github.com/MangoTeam/mockdown-client
[5] https://github.com/MangoTeam/mockdown
[6] https://github.com/MangoTeam/replication-package
[7] https://github.com/MangoTeam/auto-mock

of one of the authors [5], the three `fwt-` benchmarks are samples from the Free Web Templates collection [6], `ddg` is the DuckDuckGo search engine [62], `hn` is the HackerNews news site [3], `ace` is the Ace code editor [2], `conference` is a software-engineering conference website [4], `ieeexplore` is the full IEEE Explore website [1], which we present in simplified form in Sec. 2. The number of views in these websites range from 48 (`fwt-running`) to over 600 (`hn`), which is *over an order of magnitude larger* than layouts handled by prior work [11]. Each website was scraped at 20 different sizes with 10 designated as training examples and 10 as test examples.

Microbenchmarks. We assembled two groups of microbenchmarks: *synthetic* (repetitive, programmatically generated layouts) and *extracted* (sub-layouts of the macrobenchmarks). The extracted microbenchmarks were systematically constructed by 1) taking each macrobenchmark (a real website) and 2) extracting all its substantial sublayouts. For example, for the IEEE Xplore website, the microbenchmarks extracted are the navigation top bar, search bar, collection of authors, featured news, featured articles, upcoming conferences, and website footer. In order to test RQ3 (hierarchical scaling) we selected a subset of extracted microbenchmarks (named *hierarchical*) in which the top-level element is a one-dimensional grid, and the subelements range in complexity from 1 to 23 views, with a nesting depth of 1 to 7 levels. An array of elements is a common layout design pattern, which was present in several forms in many of our benchmarks. For example, for the IEEE Xplore website, both the collection of authors and the upcoming conferences are arrays. Mockdown scales by hierarchical decomposition and array elements are siblings in the layout hierarchy, so the hierarchical benchmarks demonstrate the impact of hierarchical decomposition on the overall performance of the algorithm.

The numbers and sizes of benchmarks in each group are listed in Tab. 2; each microbenchmark has roughly analogous complexity to the evaluation of prior work [11].

Numeric noise. Numeric noise can come from several different sources, such as approximating a rational ratio using finite-precision numbers (such as floating-point), quirks in the device-specific rendering engine, and numerical errors in floating point arithmetic. As demonstrated in our evaluation (Sec. 5.3) real-world layouts have numeric noise and cause noise-intolerant synthesis to fail.

Metrics. We evaluate the quality of inferred layouts by using the synthesized constraints to predict the placement of views at the page sizes designated for testing, and directly comparing these pixel dimensions to the original ground-truth view (which is scraped from either a browser or an Android device, depending on the dataset). We use two different metrics to quantify layout quality:

(1) *RMSD:* the root-mean-squared deviation of view corners from the original placement to our computed placement, averaged over views in the test set. RMSD captures the pixel difference between a predicted rendering and actual rendering with more weight towards outliers; for example, a rendering with RMSD under 1 is visually indistinguishable from the original, while a rendering with RMSD over 5 would look different to an average viewer.

(2) *Accuracy:* percentage of views placed within a pixel of the original view. Though used in prior work [11, 36], it can be misleading as a layout with low accuracy but low RMSD can be visually more similar to the original than one with high accuracy but high RMSD. In contrast RMSD *does* capture look-and-feel equivalence, so accuracy is mainly useful for comparison when RMSD is low.

5.2 RQ1: Applicability to Realistic Applications

For each macro- and microbenchmark, we run Mockdown on the training examples to synthesize constraints. We then pass these constraints to the Kiwi layout engine along with the page sizes from the test examples, which gives us a computed placement for each of the test examples. We compare these computed placements to the original scraped placements at the test example sizes. For the microbenchmarks we set a synthesis timeout of *2 minutes* and for the macrobenchmarks we set a timeout of *30 minutes*. Since synthesis times might vary from run to run, we ran each experiment *three times* and averaged running times.

For each benchmark, we measure the size of the benchmark (number of views) as well as the following metrics: (1) *Avg RMSD*: because RMSD is sensitive to page dimension, we average it over testing dimensions, (2) *Accuracy* as defined in the previous subsection, (3) *Synth time*: the time Mockdown takes to synthesize the layout, (4) The *number of constraints* in Mockdown's output, and (5) *Resize time*: the time for Kiwi to calculate a new placement at page resize time, averaged over the test set. This metric needs to be low because layouts are meant to be interactive and the layout would be recomputed when the user resized their window.

We performed this experiment twice, once with 3 training examples and once with 10 training examples, in order to measure the effect of the number of training examples on Mockdown. Both experiments were evaluated on 10 testing examples.

Results. The results for the macrobenchmarks are given in Tab. 1 and for microbenchmarks in Tab. 2. Overall, Mockdown successfully finds a layout for all microbenchmarks and for most macrobenchmarks, only timing out on `conference`, the 3-example `author`, and the 3-example `ace`. Moreover, independently of the number of training examples, the synthesized layouts closely match the original applications, with all RMSDs below 1. In addition, the accuracy is also relatively high: above 70% for all benchmarks except `hn`. Notice however that for `hn`, Mockdown still closely replicates the visual look of the original layout (see the low RMSD) and indeed the two layouts are almost identical to a visual inspection. It is not clear why the accuracy would be low for `hn` in particular but it could be due to rounding error from solving 6000 constraints. We omit *resize time* in the interest of space: all resize times are below 0.0004 seconds (and all but `hn` are below 0.0001 seconds), indicating that synthesized layouts can be used in an interactive setting with dynamic resizing. To sum up, we answer RQ1 **in the affirmative** – Mockdown synthesizes layouts that closely match the application for many realistic web applications.

5.3 RQ2: Noise Tolerance

To evaluate the utility of noise-tolerant local inference and the impact of our inductive biases of error and simplicity, we perform an ablation study comparing three different local inference algorithms:

`baseline`: this is a rigid Daikon-style template instantiation algorithm which instantiates invariants directly from the input data, as described in Sec. 3.

Table 1: Performance of Mockdown on macrobenchmarks.

Benchmark	Number of views	Number of Examples	Timeouts	RMS (avg)	Accuracy	Number of constraints	Synthesis time (s)
fwt-running	48	3	0%	0.26	100%	477	81
		10	0%	0.19	100%	503	40
author	56	3	100%	-	-	-	-
		10	0%	0.58	77%	626	608
ddg	57	3	0%	0.30	100%	540	41
		10	0%	0.25	100%	543	41
fwt-space	63	3	0%	0.23	99%	731	103
		10	0%	0.21	97%	748	49
ace	111	3	100%	-	-	-	-
		10	0%	0.35	90%	1047	110
fwt-main	174	3	0%	0.34	93%	1719	301
		10	0%	0.38	86%	1713	134
conference	256	3	100%	-	-	-	-
		10	100%	-	-	-	-
ieeexplore	285	3	0%	0.37	96%	2793	506
		10	0%	0.38	94%	2811	241
hn	614	3	0%	0.68	47%	6417	1689
		10	0%	0.64	50%	6430	781

Table 2: Average performance of Mockdown on each group of microbenchmarks.

Group	Number of benchmarks	Number of views (avg)	Number of examples	Timeouts	RMS (avg)	Accuracy	Number of constraints	Synthesis time (s)
Synthetic	5	5	3	0%	0.00	100%	34	9
			10	0%	0.00	100%	34	9
Extracted	55	36	3	9%	0.34	96%	127	12
			10	8%	0.33	96%	137	12

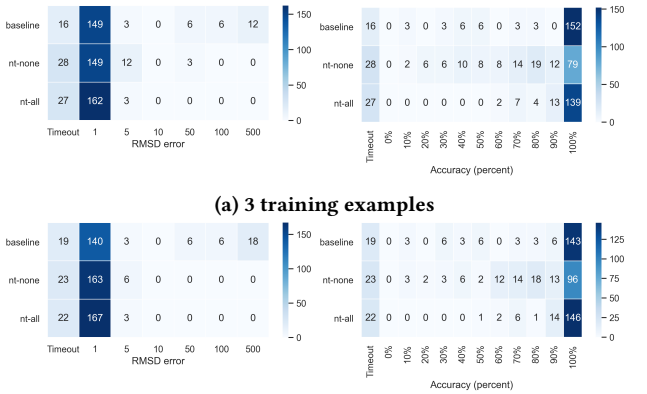
`nt-none`: this implements the noise-tolerant template instantiation but does not output confidence scores that guide global inference.

`nt-all`: this implements the full noise-tolerant inference algorithm with confidence scores based on simplicity and error.

We ran each algorithm on the full microbenchmarks set and collected (1) Synthesis time, (2) Accuracy, and (3) RMSD error, as defined above. Our experimental setup is similar to Sec. 5.2: we use 3 and 10 training examples and set a synthesis timeout of 3 minutes. We ran each benchmark three times, and our plots present all trials rather than aggregating each individual benchmark.

Results. Fig. 5 presents RMSD and accuracy, and Fig. 6 presents synthesis times for these experiments.

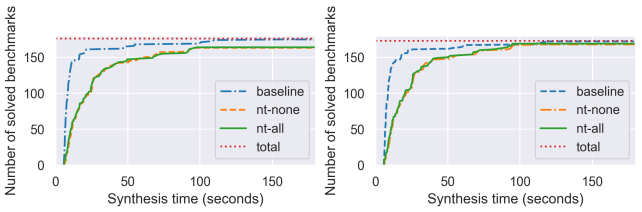
The `baseline` algorithm is brittle in the presence of numeric noise and so cannot be used for synthesis of realistic web layouts. As seen in Fig. 5a, 24 of the 192 trials (64 microbenchmarks at 3 trials each) for 3 training examples have very high RMSD error (above 10). When given 10 training examples (Fig. 5b) `baseline` does even worse, with 30 of 192 trials with a have very high RMSD error. This is because with more training samples a necessary constraint is more likely to be ruled out due to noise, leading to some anchors being completely unconstrained and thus rendered very far from their intended position. Both noise-tolerant variants, `nt-none` and `nt-all`, alleviate the brittleness and achieve low error. While `nt-none` is visually similar to the original layout, it does not consider the quality of candidate constraints, and this lack of bias in the selection of the layout invariants results in a lower accuracy than that of



(a) 3 training examples

(b) 10 training examples

Figure 5: Ablation study of Error and Accuracy for different noise-tolerance approaches.



(a) 3 training examples

(b) 10 training examples

Figure 6: Ablation study of Synthesis time for different noise-tolerance approaches.

`baseline`, as seen in Fig. 5a and Fig. 5b. The full noise-tolerant inference algorithm uses inductive bias to pick invariants, which improves the quality of the selected invariants, as demonstrated by the improvement in both accuracy and RMSD error in `nt-all`.

With regards to synthesis time (Fig. 6), noise-tolerance has an overhead, and `nt-none` and `nt-all` are slower than the native baseline algorithm. This is because the baseline algorithm infers strictly fewer candidate invariants. The difference between `nt-none` and `nt-all` is negligible, which indicates the biases do not impede performance. With more training examples, the difference between the baseline performance and the noise-tolerant performance narrows, as additional examples focus the inductive bias of noise-tolerant inference, resulting in fewer candidate invariants.

To sum up, we answer RQ2 **in the affirmative:** noise-tolerant inference is necessary for layout inference as it allows Mockdown to always find precise constraints and achieve a low error. This does come with a performance overhead, but not a large one.

5.4 RQ3: Scaling

To test how Mockdown scales with the size of the layout, we use the *hierarchical* microbenchmarks, which have a one-dimensional grid layout. For each of these grid microbenchmarks, we generate variants of different sizes by truncating the grid after n elements, with n ranging from one to ten (or the original size, whichever

is smaller). We then run two variants of Mockdown: the original version `hier` with hierarchical global learning, as described in Sec. 4 and a "flat" version `flat` that does not perform hierarchical decomposition and instead performs global inference with a single MaxSMT query. For each grid benchmark and tool variant we measure synthesis time on 4 training examples. We ran each experiment five times and average the results for each benchmark.

Results. We present the results in Fig. 7. Overall we find that `hier` outperforms `flat`, sometimes drastically. On almost half of our benchmarks (5 of 11), both variants finish quickly and so the synthesis time is comparable. This is because these benchmarks are relatively simple so decomposition does not provide a benefit. On all of the remaining 6 benchmarks `hier` performs strictly better for all sizes, scaling *linearly* in the number of elements in the array, and with the exception of `hn-posts-ten` decomposition results in tractable subproblems. Even in this case `hier` outperforms `flat` (which cannot make progress at all) – this is because each array item is itself complex and deeply nested with an element size and depth of 23 and 6. More generally `hier` reduces the synthesis task to the hardest subproblem. By contrast, `flat` attempts to solve a single global task and so becomes intractable from a certain size in 5 benchmarks. We therefore answer RQ3 **in the affirmative**: hierarchical deconstruction is crucial for synthesizing web layouts.

5.5 RQ4: Generality

In this experiment we evaluate whether Mockdown still performs well when input examples are not generated by our scraper, and more generally, are not from the web domain. To this end, we use the benchmark suite from the latest version of the layout synthesizer InferUI [36]. The suite consists of 644 Android layouts and is a subset of the popular Rico dataset [18]. For each Android application, the dataset contains layout renderings at 3 different dimensions. We evaluated Mockdown using two of these layouts as a training sample and the third as a test sample. We measured RMSD, Accuracy, Synthesis Time, and Resize Time (all as defined above) with a synthesis timeout of 3 minutes, and we ran the experiment 3 times to collect aggregate data.

Results. We present RMSD and accuracy in Fig. 8. Overall, Mockdown performs well on these benchmarks, synthesizing an indistinguishable layout (RMSD less than 5) for 545 of 644 Android apps, (and an accurate layout for 538 of them). Of the remaining 99 apps, 75 time out. There are two potential causes for this. First, the structure of apps in the InferUI dataset is flattened, so hierarchical decomposition is not applicable. Second, the 2 training examples cause local inference to produce many more candidate constraints.

We do not report a direct comparison between Mockdown and InferUI because the techniques are complementary. InferUI is designed to synthesize layouts from a single example and relies on Android-specific biases to deal with the resulting ambiguity. Mockdown on the other hand is designed to be more domain-general, and hence is unable to produce good-quality results from just one example. Running Mockdown on a single example resulted in significantly worse results than those reported for InferUI.

Nevertheless, we find it encouraging that just a single additional input example already allows Mockdown to produce highly accurate results without relying on Android-specific domain knowledge. To sum up, we answer RQ4 **in the affirmative:** Mockdown successfully synthesizes layouts for Android applications using input examples from an existing dataset.

5.6 Threats to Validity

The main threat to validity for our experiments is their origin. First, some of our microbenchmarks are generated synthetically, though we have tried to counter this by generating representations of common layout idioms (e.g., a center panel with fixed margins). Second, our macrobenchmarks and by extension the extracted set of microbenchmarks are scraped from existing websites. The numeric noise that exists in scraped layouts (e.g., noise due to rounding) represents only one source of data for Mockdown, and may not match the type of noise that will appear in layouts built by a designer in a direct manipulation tool. In principle, Mockdown's noise-tolerant local inference should be able to handle that kind of noise as well, but we leave the empirical evaluaiton of this claim to future work.

6 DISCUSSION

Unsupported layouts. Our results indicate that high errors coincide with longer synthesis times or timeouts. This is often the case in layouts that require constraints that Mockdown does not support, such as constraints for aspect-ratios or constraints for rendering text. The unsupported constraint would not be found in the local inference phase, leaving global inference to pick from many poor candidates, making its work more time consuming.

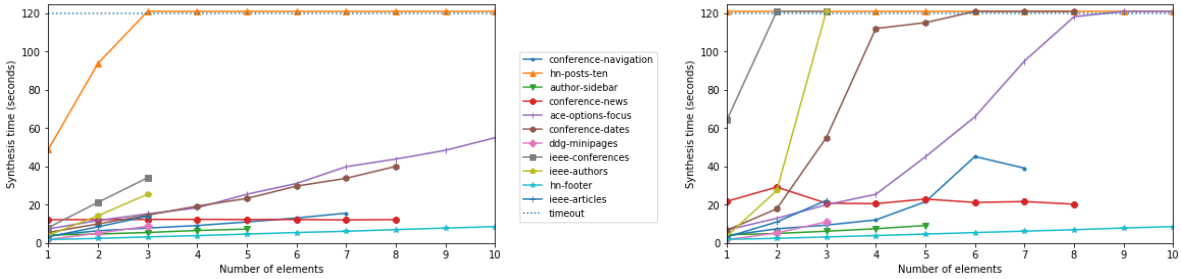
Additional examples. Unlike many works in example-based synthesis [28], Mockdown speeds up when given more examples. This is a benefit of the division of labor between local and global inference: more examples rule out more constraints at the local phase, resulting in fewer inputs to the global phase.

The cost of noise. Mockdown's noise tolerance increases the number of candidate constraints to be considered by global inference, which impacts performance. However, we believe the more accurate layouts are worth the longer synthesis times. Most realistic inputs, whether derived by scraping or direct manipulation, contain noise, making handling noisy inputs crucial.

Success of decomposition. Hierarchical decomposition proved useful in isolation, making components like grids tractable. However, when synthesizing full webpages this still may not be enough. Since hierarchical decomposition handles the relation between neighboring components, it is as efficient as its hardest subtask (*i.e.*, the view with the most direct children).

7 RELATED WORK

InferUI. The closest work to ours is InferUI [11, 36], which synthesizes constraint-based Android layouts from a single example. In the original work [11] InferUI reduces synthesis to a single SMT query that requires the layout to (1) precisely match the example and (2) satisfy a set of *robustness properties* on unseen sizes (e.g. "views do not overlap", "views fit on screen"). To improve efficiency, InferUI uses a probabilistic model learned from



(a) Hierarchical **(b) Flat**

Figure 7: Synthesis time in seconds of Hierarchical and Flat global generalization algorithms as a function of the number of elements in a grid-based layout. Lower is better.

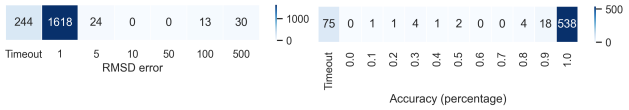


Figure 8: Mockdown's Error and Accuracy on Android layouts.

programmer-written layouts. In the followup work [36], the hand-crafted robustness properties and the probabilistic model are replaced with a learned neural model that strengthens the synthesis specification with an additional, predicted example.

At a high-level, the difference between both InferUI techniques and Mockdown is that InferUI uses a *domain-specific* bias (either hand-crafted or learned) to synthesize *simple* Android layouts (up to 20 views) from a *single* input, while Mockdown synthesizes *domain-general* and *complex* layouts (up to hundreds of views) from *multiple* inputs. InferUI has no mechanism to handle *noise* in the input examples (though their layout dataset does contain noisy inputs). Noisy data does not impact InferUI as much because when learning from one example, the synthesizer can instantiate templates with the noisy data and produce candidate constraints (albeit with slightly incorrect coefficients). In contrast, with multiple examples noisy data is problematic because the correct constraint template is rejected outright as it cannot fit the noisy data.

We consider Mockdown complementary to InferUI: our approach is more general, at the cost of requiring a few more examples from the user. An interesting direction of future work is to combine Mockdown's noise-tolerant, hierarchical inference engine with InferUI's techniques for incorporating domain-specific bias.

Visual Layout Synthesis. Other tools that assist users with creating GUIs either do not take the example-based approach [31, 34, 35, 63], focus on aspects of GUI other than the visual layout [8, 60] or generate static, vector-graphics-level layouts from images or hand drawings [10, 15, 16, 56]. Mockdown could complement the latter line of work by further generalizing the output of such tools into a resizable, constraint-based layout. A related area is synthesis of graphics programs from images [21, 42, 57], which focuses on recognizing repetitive patterns in images, and typically considers a small number of visual elements (so scalability is less of a concern).

Constraint Inference. Daikon [22, 44] has pioneered the template-based constraint inference technique that our local inference builds upon; Mockdown enhances this technique with domain-specific optimizations such as visibility-guided template instantiation and statistical parameter inference. There is a rich body of work on data-driven inference of inductive loop invariants [24, 25, 38, 43, 52, 64]. Although these techniques also infer linear relations from data, the setting is very different: invariant inference assumes abundance of data and ability to generate counter-examples, so the challenge is to efficiently find constraints that precisely fit the data, while generalization to unseen data is not an issue.

Program Synthesis. Program synthesis from examples has been used to automate tasks in many domains, such as data wrangling [20, 29, 37] and web scraping [7, 14, 32, 46]. Mockdown's global inference technique is based on *symbolic program synthesis* [33, 54, 59], which uses a constraint solver [17] to search the space of programs. MaxSAT and MaxSMT [12] solvers have been used for optimal program synthesis, e.g. in [9, 23]. Some program synthesis techniques are capable of handling *noisy examples* [19, 30, 47, 53], but most are geared towards detecting outliers (a few completely wrong examples), whereas Mockdown needs to be tolerant against a small amount of noise in each example. The only work we are aware of that handles noise distributed over a large portion of examples is [30], but this technique is specific to Version-Space Algebras and is not applicable to constraint-based layout synthesis.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their comments on the earlier drafts of this work. This work was supported by the National Science Foundation under Grants No. 1911149, 1943623, and 1955457.

REFERENCES

[1] [n.d.]. https://ieeexplore.ieee.org
[2] [n.d.]. https://ace.c9.io/
[3] [n.d.]. Hacker News. https://news.ycombinator.com/
[4] [n.d.]. ICSE 2021 Homepage. https://conf.researchr.org/home/icse-2021
[5] [n.d.]. John Sarracino. https://www.cs.cornell.edu/~jsarracino/
[6] 2020. Free Website Templates. https://freewebsitetemplates.com/. [Online; accessed 28-August-2020].
[7] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2015. Synthesizing Web Element Locators (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 331–341.

[8] Mohammad Bajammal, Davood Mazinanian, and Ali Mesbah. 2018. Generating reusable web components from mockups. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 601–611.

[9] Shraddha Barke, Rose Kunkel, Nadia Polikarpova, Eric Meinhardt, Eric Bakovic, and Leon Bergen. 2019. Constraint-based Learning of Phonological Processes. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 6175–6185.

[10] 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

[11] 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), 1–29.

[12] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ - An Optimizing SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). 194–199.

[13] Achille Brocot. 1862. *Calcul des rouages par approximation: nouvelle méthode.* A. Brocot.

[14] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 963–975.

[15] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu. 2018. From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 665–676.

[16] Alex Corrado, Avery Lamp, Brendan Walsh, Edward Aryee, Erica Yuen, George Matthews, Jen Madiedo, Jeremie Laval, LuisTorres, 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/

[17] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340.

[18] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 845–854.

[19] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 990–998.

[20] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376442

[21] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 6062–6071. http://papers.nips.cc/paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images

[22] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69, 1-3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[23] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. 2017. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*.

[24] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 69–87.

[25] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 499–512.

[26] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[27] Dawn Griffiths and David Griffiths. 2017. *Head first Android development: A brain-friendly guide.* " O'Reilly Media, Inc.".

[28] Sumit Gulwani. 2016. Programming by Examples - and its applications in Data Wrangling. In *Dependable Software Systems Engineering*, Javier Esparza, Orna Grumberg, and Salomon Sickert (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 45. IOS Press, 137–158. https://doi.org/10.3233/978-1-61499-627-9-137

[29] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. https://doi.org/10.1145/2240236.2240260

[30] Shivam Handa and Martin C. Rinard. 2020. Inductive Program Synthesis over Noisy Data. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 87âĂŞ98. https://doi.org/10.1145/3368089.3409732

[31] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14*. ACM Press, New York, New York, USA, 231–241. https://doi.org/10.1145/2642918.2647378

[32] Jeevana Priya Inala and Rishabh Singh. 2017. WebRelate: integrating web data with spreadsheets using examples. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–28.

[33] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

[34] H Kaindl, E Arnautovic, H Jelinek, T Rock, R Popp, and J Falb. 2006. Using communicative acts in interaction design specifications for automated synthesis of user interfaces. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 261–264.

[35] Michael Hanus Christof Kluß. [n.d.]. Declarative Programming of User Interfaces. *University Halle-Wittenberg Institute of Computer Science* ([n. d.]), 37.

[36] Larissa Laich, Pavol Bielik, and Martin Vechev. 2020. Guiding Program Synthesis by Learning to Generate Examples. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJl07ySKvS

[37] Vu Le and Sumit Gulwani. 2014. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

[38] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic Loop-Invariant Generation and Refinement through Selective Sampling. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. 782–792.

[39] Hakon Wium Lie and Bert Bos. 2005. *Cascading style sheets: Designing for the web, Portable Documents.* Addison-Wesley Professional.

[40] Ethan Marcotte. 2017. *Responsive web design: A book apart n 4.* Editions Eyrolles.

[41] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. Austin, TX, 51–56.

[42] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 31–44.

[43] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-Guided Approach to Finding Numerical Invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 605–615.

[44] Jeremy W Nimmer and Michael D Ernst. 2001. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electronic Notes in Theoretical Computer Science* 55, 2 (2001), 255–276.

[45] Travis E Oliphant. 2006. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[46] Adi Omari, Sharon Shoham, and Eran Yahav. 2016. Cross-supervised synthesis of web-crawlers. In *Proceedings of the 38th International Conference on Software Engineering*. 368–379.

[47] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the Enemy of Good: Best-Effort Program Synthesis. In *ECOOP*.

[48] Jorma Rissanen. 1983. A Universal Prior for Integers and Estimation by Minimum Description Length. *The Annals of Statistics* 11, 2 (1983), 416–431. http://www.jstor.org/stable/2240558

[49] Hein Rutjes. [n.d.]. Kiwi: Fast TypeScript implementation of the Cassowary constraint solving algorithm. https://github.com/IjzerenHein/kiwi.js

[50] Erica Sadun. 2013. *iOS Auto Layout Demystified.* Addison-Wesley Professional.

[51] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*.

[52] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 574–592. https://doi.org/10.1007/978-3-642-37036-6_31

[53] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Synthesizing Entity Matching Rules by Examples. *Proc. VLDB Endow.* 11, 2 (2017), 189–202.

[54] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[55] Moritz Stern. 1858. Ueber eine zahlentheoretische Funktion. *Journal für die reine und angewandte Mathematik* 1858, 55 (1858), 193–220.

[56] Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Amy J. Ko. 2018. Rewire: Interface Design Assistance from Examples. In *CHI*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174078

[57] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. https://openreview.net/forum?id=rylNH20qFQ

[58] Dan Tocchini. [n.d.]. http://gss.github.io/

[59] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 135–152.

[60] Priyan Vaithilingam and Philip J Guo. 2019. Bespoke: Interactively synthesizing custom GUIs from command-line applications by demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 563–576.

[61] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* (2020). https://doi.org/10.1038/s41592-019-0686-2

[62] Wikipedia contributors. 2020. DuckDuckGo — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=DuckDuckGo&oldid=964046557. [Online; accessed 28-June-2020].

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

[64] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 707–721.