# Responsive Designs in a Snap

Nishant Sinha
IBM Research, India
nishant.sinha@in.ibm.com

Rezwana Karim
Rutgers University, USA
rkarim@cs.rutgers.edu

## ABSTRACT

With the massive adoption of mobile devices with different form-factors, UI designers face the challenge of designing *responsive* UIs which are visually appealing across a wide range of devices. Designing responsive UIs requires a deep knowledge of HTML/CSS as well as responsive patterns - juggling through various design configurations and re-designing for multiple devices is laborious and time-consuming. We present DECOR, a recommendation tool for creating multi-device responsive UIs. Given an initial UI design, user-specified design constraints and a list of devices, DECOR provides ranked, device-specific recommendations to the designer for approval. Design space exploration involves a combinatorial explosion: we formulate it as a design repair problem and devise several design space pruning techniques to enable efficient repair. An evaluation over real-life designs shows that DECOR is able to compute the desired recommendations, involving a variety of responsive design patterns, in less than a minute.

## Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: User Interfaces, CASE

## General Terms

Algorithms, Design

## Keywords

Responsive Layout inference, Constraint-based Design, HTML, CSS

## 1. INTRODUCTION

The massive growth of mobile and tablet devices has compelled both enterprises and individual developers to create UIs viewable on multiple devices. The solution is to create *responsive* designs [31, 34], which adapt to device environments, e.g., different form-factors. They provide optimal viewing experience across devices by allowing users to read and navigate a page easily and minimizing the effort spent on window resizing, panning, and scrolling. Indeed, 2013 was named the year of responsive design[1] [16].

[1]The term *responsive* may have multiple informal meanings; here, we refer to designs whose layouts adapt to different form-factors.

*ESEC/FSE'15*, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786808

HTML and CSS are the de-facto web design languages; off-the-shelf browsers can render such designs for mobile platforms. HTML/CSS designing requires deep expertise, both for (i) encoding layouts in CSS and (ii) various layout transformation patterns that lead to responsive designs. Non-experts, instead, may prefer either *programming-by-demonstration*, *constraint-based design* or a combination of these approaches. A design *demonstration* consists of a set of UI elements or widgets laid out on a canvas, e.g., of a WYSIWYG editor [1, 11, 2]; *nested* layouts are specified using container *boxes*. Design *synthesis* engines [27, 33] are used to create renderable HTML/CSS files from the specification.

Alternatively, the designer may specify a partial UI layout on a canvas and then add *constraints* on the alignment, positioning and relative sizes of UI elements [18, 35, 26], e.g., *min-width* of a text box, *max-margin* between two adjacent elements. Constraint-based UI design has a rich history [21, 19, 18, 27, 24, 35, 26]: by enabling the designer to focus on the desired design properties instead of how to encode them, constraints reduce the layout specification burden. These constraints are finally used by a *layout engine* to *render* a device-specific design. Desktop UIs have a dedicated layout engine [10] which solves the constraints and renders design on-the-fly. In contrast, for web UIs, a browser's layout engine understands only HTML/CSS natively; hence, the page synthesis engine should transform the design constraints into CSS rules.

Creating responsive layouts which adapt smoothly to different devices is non-trivial. A naive approach to adapt UI designs to devices is by creating so-called *fluid* designs, where UI elements shrink or expand in a fixed ratio to their parent. However, such re-sizing leads to cluttered appearance and overlapping elements as devices become smaller. Instead, responsive layouts employ multiple transformations on UI elements: rearrange, resize, replace, change visibility, move elements across pages, etc. Further, to enable efficient encoding in HTML/CSS, the layouts cannot be rearranged arbitrarily: horizontal/vertical alignment of elements may change freely, e.g. rows transformed to columns, but the parent-child relationships are generally preserved across layouts.

Visual tools like Adobe Reflow [2] assist users in adapting a particular design to different form-factors using direct manipulation. However, the designer must juggle through a large number of possible designs manually, *fixing* designs through a laborious trial-and-error process. Further, the tool does not provide any layout recommendations. Grid-based CSS libraries, e.g., Bootstrap [7], allow designers to *annotate* UI elements with CSS *classes* having pre-defined responsive behaviors, e.g., a 2x2 grid will transform to 4x1 grid at the mobile form-factor. Manual annotation is laborious and error-prone: it is easy to specify wrong annotations which are hard to debug without a deep knowledge of the library and CSS

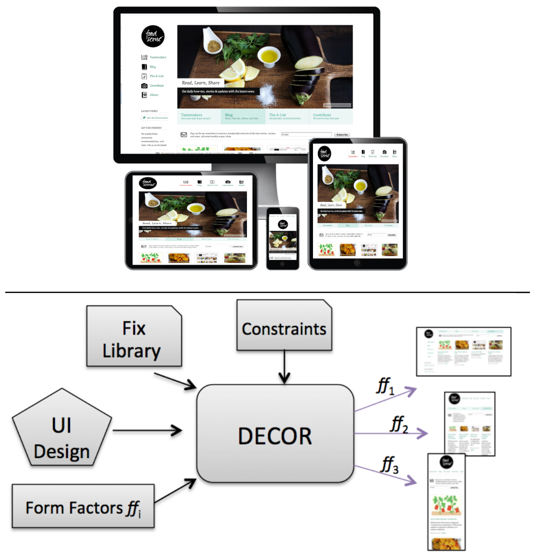


**Figure 1: (top) Real-world example of a responsive site. (below) DECOR: A responsive design recommendation tool.**

layouts.

Constraint-based design holds the promise of reducing the manual design effort. However, existing constraint-based approaches are ineffective for responsive design because they search over a restricted design space. For example, most techniques try to preserve the UI element alignments (from the original design) during adaptation and do not explore designs having grid-based rearrangments during solution discovery. Consider the original design layout in Fig. 2(top) which is adapted to the small tablet (*STP*) form-factor in Fig. 2(bottom-left). Current approaches can *shrink* the original layout to the one shown in the STP (left) area, but cannot produce the desired layout (STP (right)) which requires changing the alignment of multiple elements from horizontal to vertical. Because of their reduced search space, these techniques are forced to declare the constraint set inconsistent or drop inconsistent constraints arbitrarily [10] to keep the layout aesthetic.

We present a constraint-based design recommender tool, DECOR, to assist the designer in creating multi-device responsive UIs (Fig. 1(b)). Given an initial UI design, user-specified constraints on elements, e.g., width or margin, and a list of device form-factors, DECOR provides ranked, device-specific recommendations to the designer for approval. Intuitively, DECOR is to responsive design, as content-assist tools are to programming: the latter improve developer productivity by providing features like auto-completion and ranked suggestions, while DECOR provides design recommendations to enable rapid multi-device design. The designer may specify constraints for each form-factor and let the tool provide suggestions for each form-factor. If the designer does not approve of any suggestion, she may refine the constraints and query the tool again. To our knowledge, DECOR is the first tool to provide multiple responsive design recommendations.

The design of DECOR relies on two main observations. First, we observe that UI designs are essentially *labeled* trees, and responsive designs may be obtained by one or more *tree transformations*. Consider a design $\mathcal{D}$ with the design tree $T$, containing an UI element node $n$ with children $c_1$ and $c_2$ laid out horizontally on being rendered ($n$ is a *HBox*). Suppose $c_1$ and $c_2$ appear too narrow on the tablet device. We can fix $\mathcal{D}$ by transforming the orientation of $n$ to *VBox* where $c_1$ and $c_2$ align vertically and hence appear sufficiently wide. We formalize responsive design as a *design repair* problem: given a UI design, and a set of *design* constraints $C$, compute the set of *fixes*, so that the fixed design satisfies $C$ for each desired device. The notion of a *fix* is at the core of our approach: a fix effectively characterizes the designer's actions as she goes about adapting the design for different devices.

Our second observation is that although the total number of potential tree transformations (space of designs) is huge, only a few transformation patterns are used in practice. However, trying out all permutations of even this smaller set of transformations on real UI design trees leads to a combinatorial explosion. To build a practical tool, we devise several pruning heuristics which exploit the independence of fixes, similar to partial order reduction methods [25, 23], and bias the search towards more general and *ordered* fix sequences to prune the search space (cf. Sec. 5).

We evaluate DECOR over a representative set of synthetic and real-life UI designs. Our results show that DECOR is able to compute several design recommendations, involving a variety of responsive design patterns, in less than a minute. The recommendations obtained are realistic, i.e., they precisely mimic the responsive behaviors created manually by experienced designers of real-life responsive sites. In some cases, DECOR is also able to suggest alternative design repairs which require fewer fixes as compared to the original responsive site. The paper makes the following contributions.

- We formalize the responsive design problem as a constraint-based repair problem over labeled design trees. We present a new specification language to specify constraints over UI elements for multi-device designs.
- We model the space of design transformations using a *design tree graph* (DTG), whose nodes are design trees and transitions are repairs. An algorithm is presented to search over a DTG systematically to find designs satisfying constraints.
- We present efficient pruning heuristics to avoid combinatorial explosion during DTG exploration by avoiding redundant fixes, exploiting mutual independence of fixes and enforcing a tree-based ordering on fix sequences.
- We describe an implementation and an evaluation of DECOR, a tool for creating multi-device responsive designs. We demonstrate that DECOR is able to provide realistic recommendations for real-life designs within a minute. We conduct a study to compare the user experience with DECOR against a commercial responsive design tool.

## 2. OVERVIEW OF THE APPROACH

Consider the UI design $\mathcal{D}$ shown in Fig. 2(top-left), based on a real-life responsive website [6]. Suppose designer uses a WYSIWYG design tool [2] or a mockup builder [3, 9] to create this design for the desktop form-factor (width > 1200px). Now, she wishes to adapt $\mathcal{D}$ to be visually appealing for the following devices (form-factors): (i) tablet landscape ($width$ < 1024px) (ii) small tablet portrait ($width$ < 600px) and (iii) mobile-portrait ($width$ < 320px). Instead of manually tweaking $\mathcal{D}$ for each form factor, the designer uses DECOR to obtain desired designs. First, she provides a set of *design* constraints on attributes of UI elements, e.g., min-width, min-margin, font-size or their combinations (cf. Sec. 4.1). DECOR then generates *ranked* design recommendations for each desired form-factor, that satisfy the above constraints. Now, the designer picks one of the designs if she approves of it; otherwise, she adds more constraints and asks DECOR to come up with better recommendations. Once the designer approves all the designs,

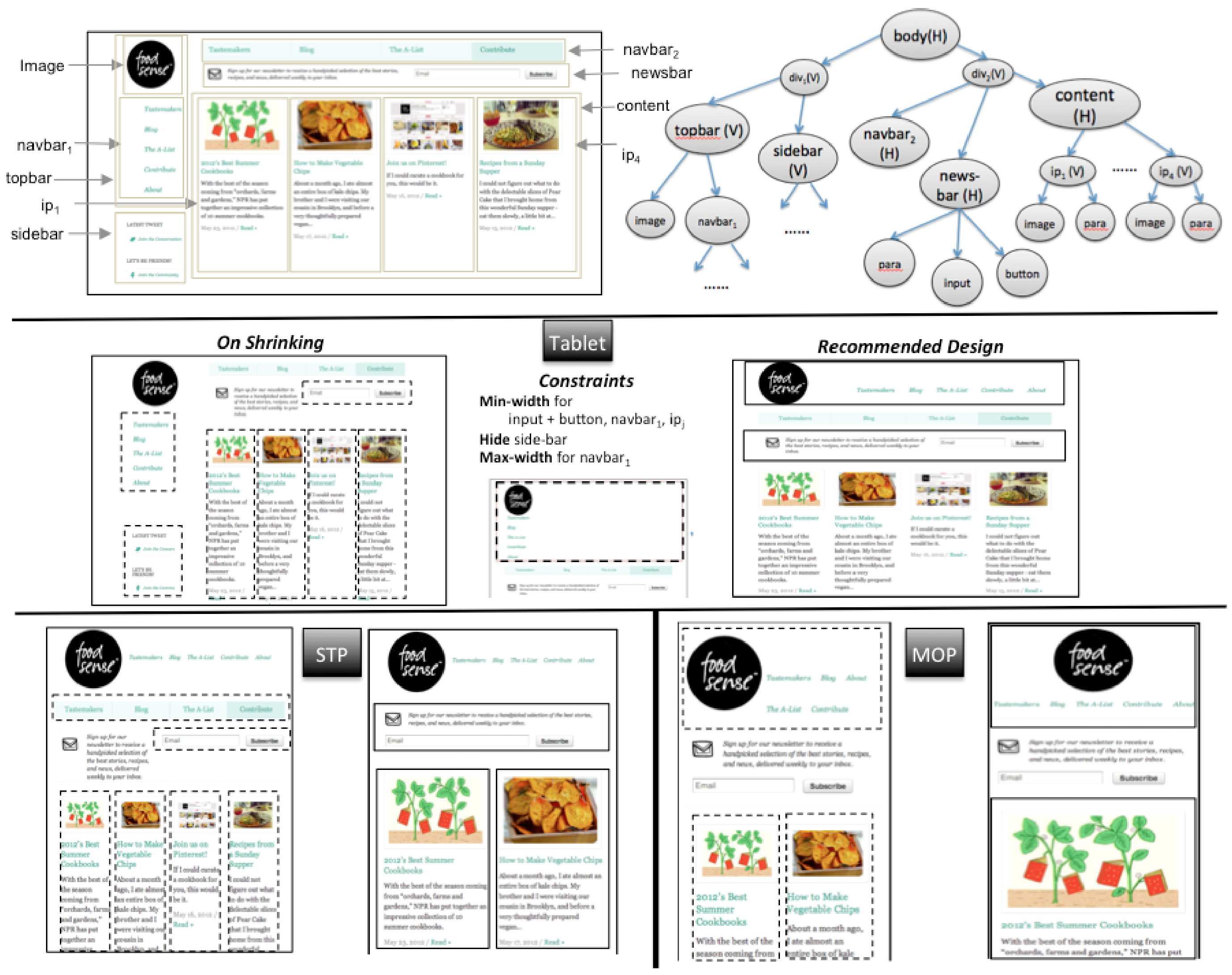


**Figure 2: An illustrative responsive design example, for tablet, small tablet portrait (STP) and mobile portrait (MOP) form factors. Dotted borders mark visually unappealing UI elements, solid borders show the fixed versions.**

DECOR generates faithful HTML/CSS files encoding the designs, which may be viewed in any off-the-shelf mobile/desktop browser and responds to form factor changes.

To adapt the original design $\mathcal{D}$, the tool first extracts a hierarchical, annotated, design tree (DT) $T$ from $\mathcal{D}$ (cf. Sec. 3 for details), shown in Fig. 2(top-right), and then *repairs* $T$ to satisfy all constraints for each form-factor (FF) $f$. Repairing $T$ for each $f$ involves computing one or more sequence of *fixes*, called *fix-chains*. Each fix-chain transforms $T$ to a visually appealing tree $T'$ for $f$. These fix-chains are then ranked and the designs encoding the repaired trees are shown as recommendations.

We describe how DECOR works for the tablet-landscape (*tl*) FF. DECOR starts with most obvious repair: it *shrinks* the original design (preserving relative sizes of elements) to the width of *tl* FF. This design is not visually pleasant, e.g., the following elements appear too *narrow*: email *input* and subscribe *button* under the newsbar, the navbar$_1$, sidebar and the four ip$_i$ ($1 \leq i \leq 4$) elements (Fig. 2, Tablet). To eliminate these issues, the designer specifies constraints: (a) min-width(navbar$_1$) = 200px, which applies to navbar$_1$ in all FFs. Further, she determines that the sidebar should now be invisible and specifies constraint (b) hidden(sidebar) for all FFs less than *tl*. Similarly, (c) min-width constraints for *input*, *button* and $ip_i$ elements are specified.

Now, DECOR uses these constraints to fix the design: it applies fixes from its *fix library* (cf. Sec. 4.2) to repair the design. Here, the computed fix-chain has 3 fixes. First, the fix *hide-element* is applied on the sidebar. Then, the fix *reduce column* (red-col) is applied on the *body* element: *body* has two columns ($div_1$ and $div_2$ boxes) in the original design, this fix converts *body* into a single column, i.e., it *stacks* $div_1$ over $div_2$. The resulting design is shown in Fig. 2, Tablet (middle): elements *fs-image* and *navbar*$_1$ appear too wide again, making the design unpleasant. The designer specifies a *max-width* for *navbar*$_1$; the tool then applies the fix *red-row* to obtain the desired design (Fig. 2, Tablet (right)). Note how the min-width constraints for *input* and *button* elements (inside *newsbar*) are satisfied, without applying fix directly on them; fixing the *higher* (closer to root) tree nodes often obviates the need for fixing specific *lower* nodes and leads to short fixes.

For small tablet portrait (stp) FF, DECOR again shrinks the previous design to *stp*'s width (Fig. 2(STP)). Now, the unpleasant elements are: navbar$_2$, email, subscribe and ip$_i$. Suppose the designer decides to hide navbar$_2$ for stp and adds constraints for above elements. Now, DECOR tries to satisfy all given constraints for stp: it applies *red-col* on newsbar (move input and button to a new row) and *red-col* on content, i.e., convert from 4-column, 1-row box to 2-column, 2-row box, to obtain the desired design for stp (STP(right)). On shrinking this design to mobile portrait (mop) FF (Fig. 2(MOP)), navbar$_1$ and ip$_i$ appear squished. DECOR applies *red-col* on both topbar (navbar$_1$ gets a new row) and content (switch to 1-column, 4-rows) to repair this design.

**Discussion.** Note that a designer may not conceive of all constraints upfront; she may need to add constraints *interactively* and re-run the tool multiple times. Constraints and fixes are *not* related one-to-one: a single constraint may determine fixes for *multiple FFs*, which saves the designer from re-applying the multiple fixes manually. To fix designs manually without constraints, the designer must be able to visualize the design hierarchy and various fix interactions mentally and adapt the design by trial-and-error, which becomes unduly hard with complex designs and longer fix-chains.

## 3. PRELIMINARIES

**Designing UIs.** Static UI design involves defining the UI *elements*, the *layout*, e.g., left-to-right alignment of elements, and the *styling*, e.g., color, font-size of elements. Either direct manipulation tools, e.g., Adobe Photoshop, or UI programming frameworks [8, 5] or a combination of them [1, 10] may be used. For creating web UIs, HTML [8] and CSS [4, 5] are the de-facto languages (JavaScript is used for the dynamic behavior). Web UI programming involves a significant learning curve: most direct manipulation tools do not generate good-quality code and end-users or unskilled designers struggle to encode their designs in HTML/CSS [28, 33].

**Devices and Form-Factors.** Recently, a variety of mobile devices have become available. These devices are distinguished by their display *form-factors*, i.e., width-height ratio (in pixels), denoted as *width***x***height*) Devices may be categorized into following major categories: desktop, tablet, phone having widths 1200px or more, 500px - 800px and 320px - 480px, respectively. Further, phones and tablets typically support both *portrait* and *landscape* modes.

**Breakpoints, Viewports.** A *breakpoint* is, intuitively, a transition point for the design and is specified by its width, e.g., 700px; transition points affect noticeable discrete visual changes in the design (cf. Fig. 1(a)). A designer picks breakpoints in order to ensure visual appeal on a small subset of devices: each breakpoint *bp* corresponds to one or more devices having width close to *bp*. In this paper, we use the term *device* or *form-factor* to specify device properties, *viewport* to denote the design container or renderer, e.g., the browser, and *breakpoint* to specify the viewport width at which the design changes. Viewport may correspond either to the entire device screen or the reduced width of a resized browser.

**Responsive designs.** A declarative specification in HTML/CSS, may be *rendered* on different devices or viewports in completely different ways. Although the underlying structure (HTML) remains same, the layout, positioning, styling and visibility of elements may be changed via CSS3 rules specific to devices or viewports. These rules can be specified using *media-query* directives [12]: based on a combination of device and viewport properties, different CSS rules may be triggered. For the example in Fig. 2, we write

```
media (min-width: 1200px) { #sidebar {display: block;} }
media (min-width: 1024px) and (max-width: 1200px) {
    #sidebar {display: none;}
}
```

to make *sidebar* element visible for viewports of width $> 1200$px and hide it for lower width viewports. Media queries can be complex Boolean combinations of various device properties [12] and allow designs to adapt to devices in non-trivial ways.

**UI Designs Formally.** A design $\mathcal{D}$ consists of a collection of UI elements $\mathcal{B}$ (basic widgets, containers), represented as rectangular objects (also called *boxes*). Non-rectangular UI objects, e.g., images, are represented by their bounding boxes.

**Design tree (DT).** We represent a design formally using a *design tree* $T = (N, ch, orn, \sigma)$ which consists of a set of nodes $N$, and maps $ch$, $orn$ and $\sigma$. We use $root(T)$ and $leaves(T)$ to represent the root node and the set of leaves of $T$, respectively. Given a node $n \in N$, $ch(n)$ denotes the *ordered* list of children of $n$ in $T$. The leaves of $T$ correspond to basic UI widgets, e.g., button, selection, image, section, and intermediate nodes are containers. Functions $orn$ and $\sigma$ capture the design features of $T$: $orn(n)$ denotes the layout of children of $n$ and $\sigma(n)$ denotes the styling, positioning and other attributes of $n$, as explained below.

**Styling.** Each node $n \in N$ is labeled with a set of attributes which determine the *size*, position and visual appearance of $n$ on being rendered. Given a set of attributes $\mathcal{A}$ drawing values from set $\mathcal{V}_{\mathcal{A}}$, the attribute map $\sigma : N \to \mathcal{A} \to \mathcal{V}_{\mathcal{A}}$, which maps a node $n$ and an attribute $a \in \mathcal{A}$ to a value $v \in \mathcal{V}_{\mathcal{A}}$. Attribute values may be specified either using absolute measures, e.g., in pixels (px) or relative to parent of $n$, using percentages. Percentage values allow the design to be *fluid*. For example, suppose a child node $n$ has width $40\%$ of its parent, the root element $r$. Now, if we reduce the (rendered) width of $r$, $n$'s width will also *shrink* in the same ratio. Although responsive designs are fluid in general, fluidity is not enough: elements typically are rearranged to adapt to devices.

**Hierarchical Layouts: Grids, HVBoxes.** A DT is rendered as a nested collection of boxes; different layouts are obtained by aligning children boxes *vertically* or *horizontally* inside the parent (container) box. We specify the orientation of each non-leaf node $p$ using a generic *grid box* primitive; $orn(p) = (k, l)$ means that the children of $p$ are aligned into a grid with $k$ rows and $l$ columns, and $p$ is said to be a $(k, l)$-box. When $k = 1$ ($l = 1$) we say that $p$ is a *HBox* (*VBox*), i.e., contains a single row (column) of elements. As we will see later, the grid box notation allows us to capture layout transformations conveniently.

**Rendering Design Trees.** A DT $T$ may be visualized using a suitable renderer, e.g., an off-the-shelf browser. The renderer interprets $T$ and projects it on to a 2-D plane, as a *flat* design $\mathcal{D} = (\mathcal{B}, \sigma_r)$, where $\mathcal{B}$ is a set of visible UI boxes corresponding to nodes in $T$ and the attribute map $\sigma_r$ is defined for each $b \in \mathcal{B}$. For a box $b_n \in \mathcal{B}$ corresponding to $n \in N$, $\sigma_r(b_n) = \sigma(n)$ and $\sigma_r(b_n)(p)$ is defined in absolute pixel values for all position and size attributes $p$, e.g., left, top, height, width. Given a device of form-factor $f$ = W**x**H, we say that a tree $T$ *fits* the device if the rendered box $b_{root}$ for $root(T)$ is narrower than $W$, i.e., $\sigma_r(b_{root}, width) < W$.

## 4. CONSTRAINT-BASED DESIGN REPAIR

Given an initial DT $T$ drawn for a breakpoint *bp* (e.g., 1200x768), we want to transform $T$ into a new tree $T'$ which is visually appealing for a lower-width breakpoint, e.g., 800x600. Once $T'$ is obtained, we can encode $T$ and $T'$ together into HTML/CSS automatically using media queries and rules presented in [33]. We can obtain $T'$ by *shrinking* the width of all elements by a fixed ratio; however, this may lead to visually unpleasant designs, e.g., text may be unreadable, page elements may overlap or appear cluttered.

To create a repair strategy, we first must define what a *good-looking* design is. While it is hard, if not impossible, to quantify visual appeal precisely, we can often characterize design deficiencies

| | | | |
|---|---|---|---|
| Constants | | $k \in \{\mathbb{N}$, *true, false, strings, numbers, ..*$\}$ | |
| Tree nodes | | $n$ | |
| Node attributes | | $a \in A$: {*width, height, margin, min-width, font-size, ..*} | |
| Operators | | $\oplus$: $\{=, <, >, \leq, \geq, ..\}$ | |
| Breakpoints | | $bp \in B$: {1024x768, 320x480, tab, stp, mop, all,...} | |
| Constraints $C$ | ::= | [ *bpl* ] $\phi$ | |
| *bpl* | ::= | $bp(, bp)^*$ | |
| $\phi$ | ::= | $(L \mid N) \mid \phi \vee \phi \mid \phi \wedge \phi$ | |
| $L$ | ::= | $Cx \oplus k$ | |
| $Cx$ | ::= | $n.a \mid \sum(k * Cx)$ | $k \in \mathbb{N}$ |
| $N$ | ::= | hidden$(n)$ \| halign$(n_1, n_2)$ \| valign$(n_1, n_2)$ | |

**Table 1: Constraint language for multi-device designs.** $L$ **= linear predicate,** $Cx$ **= constraint expression. Expressions should be type-correct.** $bp$ **=** *all* **denotes the set of all breakpoints.**

in terms of constraints on size, spacing and other attributes, e.g., a design is not looking good because an image is too wide, or a paragraph text is too narrow. Therefore, we ask the designer to supply a set of *design constraints* on UI elements: a design satisfying these constraints is assumed to be visually appealing.

Given a set of constraints $\mathcal{C}$, we can repair $T$ in large number of ways to satisfy $\mathcal{C}$, e.g., we may fix the size attributes of nodes or change their orientation or move around subtrees in $T$. Our key observation is that designers rely only on a small subset (about 10) of these fixes for creating responsive designs, e.g., converting rows to columns or vice versa, changing margin sizes, fonts and hiding optional controls. These patterns are applied carefully only to a selected set of tree nodes, and are in many cases independent of each other. These observations allow to devise a systematic approach to search over the set of possible fixes efficiently. Our constraint-based repair framework is *extensible*: both constraints and fixes can be added, enabled and removed on demand.

## 4.1 Design Constraints

Formally, design constraints are logical predicates on properties of tree nodes, e.g., attributes (width, margin), orientation and the parent-child relationships, e.g. for a paragraph UI element $p$, $p$.width $\leq 600px$ is an instance of a *max-width* constraint. Constraints may refer to multiple elements simultaneously, e.g., $n_1$.width $+ n_2$.width $\leq 200px$, for nodes $n_1$ and $n_2$.

Table 1 shows our constraint language formally. The language is inspired by, and generalizes, the constrained CSS language [19], which allows specifying linear constraint rules in addition to CSS rules. To design for multiple-devices, each constraint may be specified for one or more breakpoints *bpl*. For example, ($[all]n1$.width $>$ 300px) constrains the min-width of $n_1$ to be 300px for all breakpoints. *Named* predicates (N) allow specifying additional constraints: hidden$(n)$ constraints $n$ to be invisible, halign$(n_1, n_2)$ (valign) constrains sibling nodes $n_1$ and $n_2$ to be horizontally (vertically) aligned.

**Trees satisfying constraints.** A DT $T$ is said to satisfy a constraint $\phi$ (denoted $T \models \phi$) iff $\phi$ evaluates to $true$ given the node attribute values in $T$. Given constraints $\mathcal{C}$, we say that $T$ satisfies $\mathcal{C}$ for a breakpoint *bp* ($T \models_{bp} \mathcal{C}$), iff $T \models \phi$ for each constraint of form $([.., bp, ..]\phi) \in \mathcal{C}$. Our goal is to find *all* trees $T \models_{bp} \mathcal{C}$ for each breakpoint *bp* and allow the designer to select the desired one.

## 4.2 Design Fixes

A *fix* is the basic unit of design tree transformation. Given a DT $T = (N, ch, orn, \sigma)$, a fix may modify either (i) the set of nodes $N$ (insert or delete nodes), or (ii) the ordering of children $ch$, or (iii) the orientation $orn$, or (iv) one or more attributes (height, width, margin, font-size, ...) of a node $n \in N$. Multiple fixes may be required to repair a $T$, i.e., satisfy constraints $\mathcal{C}$.

| Fix Name | Description | Fix | Description |
|---|---|---|---|
| red-margin | Reduce Margin | res-child | Modify Children Width Ratio |
| red-row | Reduce Rows | red-col | Reduce Columns |
| reorder-child | Reorder Children | nav-to-select | Replace `list` by `select` |
| add/del-node | Delete/Add Node | mod-line | Change line-height |
| mod-img | Change Image Size | mod-txt | Change Text Properties |

**Table 2: List of common fixes.**

Table 2 shows common fixes used in responsive designs [15, 31]. For example, applying *red-row* fix at node $n$, $orn(n) = (2, 2)$ (2x2 grid box, cf. Sec. **??**) reduces the number of rows of $n$, e.g., $orn'(n) = (1, 4)$ (1x4 grid box) whereas *red-col* reduces the columns (increases rows) of $n$ e.g., $orn'(n) = (4, 1)$ (4x1 grid box). Each fix targets one or more types of constraint violation distinctly, e.g., *red-margin* (reduce margin values) targets the constraints over min-width or max-margin of elements. DT nodes are not explicitly added/removed: their *visible* attribute is toggled.

DECoR takes a *fix library* $F$ as input and uses fixes from $F$ to repair $T$. A *combination* of these fixes gives rise to popular responsive patterns [15, 31]. For example, the *columnflip* and *mondrian* patterns [31, 15] can be captured by *red-col* and *red-row* fixes. Similarly, *basic gallery*, *column drop* or *column flip* patterns [15, 31], are different instances of the *red-col* fix depending on the number of elements in the grid.

Formally, a fix is a rewrite rule at a node $n$ of $T$:

$$(n, [c_1, \cdots, c_m], ch, orn, \sigma) \Rightarrow (n, [c_1, \cdots, c_m], ch', orn', \sigma')$$

where $[c_1, \cdots, c_m]$) are the child nodes of $n$ in $T$ and functions $ch$, $orn$ and $\sigma$ ($ch'$, $orn'$, $\sigma'$) denote the children, orientation and attributes, resp., for $T$ before (after) transformation.

We describe the fixes *red-margin* and *red-col* informally here. Detailed rules for fixes in Table 2 are omitted to the full version of the paper [14]. Recall that $n$ is rendered into 3 nested boxes: margin, border and content boxes. The *red-margin* fix on $n$ increases $n$'s content box width and decreases left/right margins of $n$, without changing the margin box width of $n$. It first computes the margin value $\gamma$ that can be deducted from $n$'s current margin value without violating the min-margin constraints on $n$ in $\mathcal{C}$ (if they exist). Both left and right margins are then reduced by $\gamma$; in turn, the width of content-box of $n$ is increased by the total reduced margins. This *expands* the contents of $n$ (and its children) which now may satisfy a min-width constraint. Note how a single fix may satisfy/falsify multiple constraints (min-margin, min-width).

The *red-col* fix applies to most designs. Suppose $n$ is a $(k, l)$-box with $m$ children $c_1, \ldots, c_m$; *red-col* reduces the number of columns $l$ of $n$ (increases $k$, in turn). Fig. 3 shows two *red-col* fixes applied in sequence ($(k, l) = (1, 4) \rightarrow (2, 2) \rightarrow (4, 1)$) on a design extracted from Fig. 2. The fix transforms $n$ into a $(k', l')$-box, where the columns $l'$ is the *largest factor* of $m$ less than $l$. The new width and margin values for children $c_i$ are as follows: (**margins:**) the top (bottom) margin for each $c_i$ is set to the maximum of a predefined *global-min-margin* value and their left (right) margin values in previous $(k, l)$-box. For the new rows added to box of $n$, we compute an average horizontal (left/right) margin value from the previous left and right margin values of each child. In case the new margin value is lower than *min-margin* constraint $\gamma_i$ for $c_i$ then we set it to $\gamma_i$. (**widths:**) For updating width of each $c_i$, *red-col* uses a simple heuristic: the *margin-box* width for each $c_i$ is obtained by dividing the width of $n$ among each child $c_i$ in a row of $(k', l')$ using their relative width ratio. This relative width ratio is computed from $c_i$'s width in $(k, l)$. Finally, the width of *content box* of $c_i$ is obtained by deducting already computed left/right margin values for

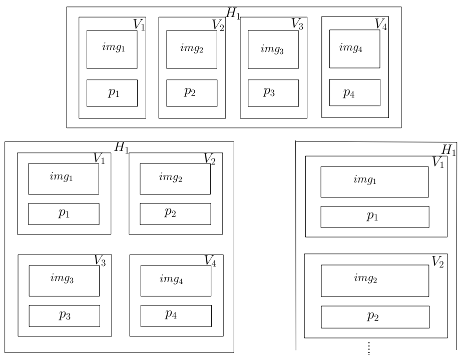


**Figure 3: Illustration of the 4-2-1 responsive pattern.**

each $c_i$ from corresponding margin-box width. Other heuristics for margin and width computation are also possible; after fix is applied, the designer can further fine-tune these values.

We have designed most fixes to be *local*, i.e., a fix on $n$ keeps its margin box fixed and hence does not affect its non-subtree nodes in the tree. In general, fixes and constraints interact in complex ways; applying a fix may lead to satisfaction of some constraints and violation of others. In other words, fixes may *enable* or *disable* other fixes. Therefore, different *permutations* of fixes may lead to different designs. To ensure we obtain multiple satisfying DTs, we need to explore several permutations of multiple fixes. Because the number of such fix sequences is huge, we introduce *design tree graphs* to explore *representative* fix sequences systematically.

## 5. DESIGN TREE GRAPHS

A **Design Tree Graph** (DTG) $G = (Q, \iota, \Sigma, R, \Psi, \mathcal{T})$ over a fix library $F$ and constraints $\mathcal{C}$, consists of a set of states $Q$, set of initial states $\iota$, a set of transition labels $\Sigma$, transition relation $R \subseteq Q \times \Sigma \times Q$ and a mapping function $\mathcal{T}(\cdot)$, where $\mathcal{T}(q)$ is the DT $T$ corresponding to state $q$. The set $\Psi \subseteq Q$ consists of *good* states $q$ with satisfying trees, i.e., $\mathcal{T}(q) \models \mathcal{C}$. Each transition label $g \in \Sigma$ is a pair $g = (n, f)$ where $n$ is a node in a DT and $f \in F$. Given a state $q$, DT $T = \mathcal{T}(q)$ and label $g = (n, f)$ where $n \in \mathcal{T}(q)$ and $f \in F$, transition $(q, g, q') \in R$ iff applying $f$ to node $n$ in $T$ results in $T'$ and $\mathcal{T}(q') = T'$. The outgoing edges of a state $q$ in $G$ represent the set of *valid* fixes (described below) which may be applied to $\mathcal{T}(q)$. We construct a DTG separately for each breakpoint; each initial state corresponds to a $T$ to be repaired.

Finding responsive designs boils down to exploring paths of $G$ to find a good state. Alg. 1 shows the algorithm for constructing a DTG and finding all *good* fix-chains, which lead to good states. For a breakpoint *bp*, the algorithm starts with a satisfying tree $T_0$ from the previous breakpoint, *shrunk* to *bp*'s width. GETVALIDFIXES computes the set of *valid* node-fix pairs $\Sigma' \subset \Sigma$ at a state $q$ over library $F$ (cf. Sec. 6). Each node-fix pair in $\Sigma'$ is applied iteratively (using procedure APPLYFIX) to obtain new tree $T'$ and state $q'$. A DTG $G$ may contain cycles: the algorithm uses a state matching criterion to detect cycles and avoid loopy fix-chains.

**Example.** Recall the design $\mathcal{D}$ for the 4-2-1 pattern shown in Fig. 3. We now describe how EXPLOREDTG algorithm constructs the DTG for $\mathcal{D}$ for *stp* breakpoint, shown in Fig. 4. The algorithm starts with the initial state containing the DT $T_0$ for $\mathcal{D}$ ($H_1$ is a $(1, 4)$-box). Suppose after shrinking to *stp*, the width of $img_i$ in $T_0$ violates $\mathcal{C}$. Fixing $img_i$ directly may not be useful; instead we may need to fix one of its ancestors. Let us define the *bad nodes* in a DT to be all nodes whose some descendant violates $\mathcal{C}$. Here, bad nodes in $T_0$ are $\{img_i, V_i, H_1\}$.

```
Input: Breakpoint $bp$, Initial Design Tree $T_0$, Constraints $\mathcal{C}$, Fix Library $F$
Output: Set $\Lambda$ of satisfying fix-chains $\lambda$
$\mathcal{T} := \{(q_0, T_0)\}$; $R := \Psi := \emptyset$; $Q := \{q_0\}$; $G := (Q, q_0, \Sigma, R, \Psi, \mathcal{T})$;
$\Lambda := \emptyset$ ;
EXPLOREDTG ($q_0$, $\epsilon$)

EXPLOREDTG ($q$, $\lambda$) // $\lambda$ is current fix chain
$T := \mathcal{T}(q)$
if $T \models_{bp} \mathcal{C}$) then  $\Lambda \cup = \{\lambda\}$; $\Psi \cup = \{q\}$ return
$\Sigma'$ := GETVALIDFIXES ($T$, $F$) // $\Sigma'$ is an ordered list of pairs $(n, f)$
if $\Sigma'$ is empty then  return //Backtrack
foreach $g \in \Sigma'$ do
    $T'$ := APPLYFIX($g$, $T$)
    Create fresh $q'$ where $\mathcal{T}(q') = T'$
    $\lambda' = (\lambda \rightarrow g)$; $Q \cup = \{q'\}$; $R \cup = \{(q, g, q')\}$
    EXPLOREDTG ($q'$, $\lambda'$)
```

**Algorithm 1:** Computing all satisfying fix-chains.

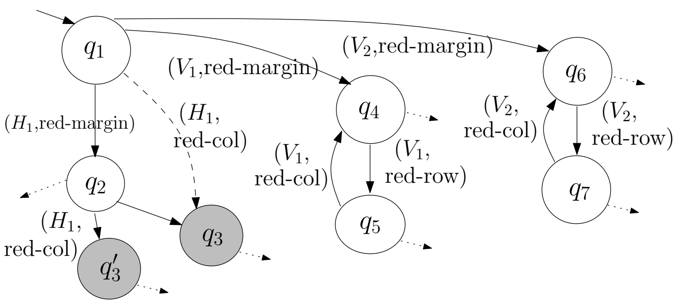


**Figure 4: A snippet of the DTG for fixing the original design (4-columns) for 4-2-1 pattern, shown in Fig. 3.**

Suppose EXPLOREDTG picks a bad node $V_1$ to fix and applies *red-margin* to $V_1$ (state $q_4$); still $img_i$ is too narrow. The algorithm continues to apply fixes *red-row* (which makes $V_1$ single row and shrinks $img_i$ further), followed by *red-col* to $V_1$. At this point, it detects a cycle and backtracks without success. Suppose after trying to fix $V_2$, $V_3$ and $V_4$ similarly, the algorithm backtracks to $q_1$ without success. Now EXPLOREDTG picks node $H_1$ and applies fix *red-margin* to $H_1$ to reach another bad state $q_2$. The *red-col* fix is applied to $H_1$, converting it to a $(2, 2)$-box at state $q_3'$ (tree $T_3'$). Because the image widths now satisfy $\mathcal{C}$, we obtain a good fix-chain ($H_1$, *red-margin*)→($H_1$, *red-col*) for *stp*. The algorithm will also find a shorter chain ($H_1$, *red-col*) for *stp* leading to tree $T_3$ at $q_3$. After finishing with *stp*, EXPLOREDTG computes fix-chains for the next breakpoint *mop*, starting from either $T_3$ or $T_3'$. A satisfying DT for *mop* breakpoint is obtained by applying *red-col* to $H_1$ again, resulting in all $V_i$'s being aligned vertically.

EXPLOREDTG is inefficient in multiple ways. (1) The algorithm tries to fix $V_i$'s unnecessarily. Although we cannot rule out fixing $V_i$'s statically (*red-margin* may be a valid fix), we can avoid *red-row* for $V_i$ because applying *red-row* will not satisfy the min-width constraint. Similarly, we can avoid the large number of fix permutations on $V_i$'s because they are tree siblings and can be fixed independent of each other. (2) The algorithm may try to both fix $V_i$ followed by $H_1$ and vice versa. In most cases, fixing nodes in the tree order ($H_1$ followed by $V_i$) is sufficient. We now present a set of optimizations to explore DTGs more efficiently.

## 6. EFFICIENT EXPLORATION OF DTGS

Given a tree $T$ with $N$ nodes and fix library $F$, the number of fix-chains in the corresponding DTG is $O(N! \cdot p^N)$ where $p = (|\,F\,| + 1)$. We present path-pruning heuristics to explore a small

```
Input: Design Tree $T$, Fix Library $F$,
Constraints $\mathcal{C}$
Output: Fix List $\Sigma'$
GETVALIDFIXES ($T$, $F$)
$B$ := nodes from $T$ violating $C$; $G$ = []
Sort $B$ in decreasing tree height order
foreach $n \in B$ do
    $F'$ := GETVALIDNODEFIXES($n$, $F$)
    $\Sigma' = \Sigma' \cdot [(n, f), f \in F']$
return $\Sigma'$
```
(a)

| Fix | Node Properties |
|---|---|
| red-col | VBox |
| red-row | HBox or Root |
| red-margin | No/Min. margin |
| nav-to-select | Non-navbar |
| red-font | No descendant with text |

(b)

**Figure 5: (a) The algorithm GETVALIDFIXES (b) A list of invalid fixes and their target node properties.**

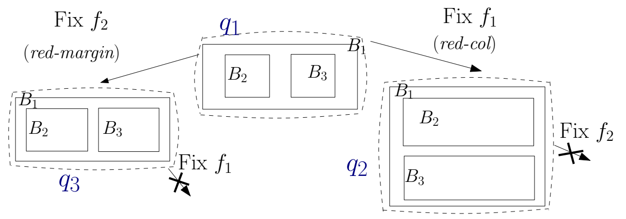


**Figure 6: A DTS fragment illustrating *dependent* fixes.**

*representative* set of fix-chains in DTG based on (i) pruning the fixes applied to each node, and (ii) selecting the order of nodes to fix. The latter may omit some satisfying DTs from exploration; however, in our evaluation, no desired DTs were eliminated.

**Valid and Enabled Fixes.** Given a DTG state $q$ with tree $T$, the set of *valid* fixes at $q$ consists of fixes which may be applied to some node in $T$ because their pre-condition is satisfied. In general, many fixes are *invalid*, e.g., we cannot apply *red-col* on a *VBox* element, or *red-margin* on a node with no margin, or reduce font on non-text nodes. For each node $n$, the procedure GETVALIDNODEFIXES computes only the valid fixes (Tab. 5) and prunes away the rest (e.g., *red-row* in Fig. 4). The procedure GETVALIDFIXES combines these fixes for the bad nodes in $T$ and ranks them to obtain enabled fixes $\Sigma'$ for $q$.

**Redundant fix pruning.** We can prune the valid fix set further by removing fixes which may not help satisfy any violated constraint in $\mathcal{C}$. We detect them by relating the constraints violated and the bad nodes in $n$'s subtree as follows.

- If $n$ is a bad node but none of descendants are bad, then fixes *red-col* and *red-row* are redundant. This is because both these fixes modify only $n$'s descendants, not $n$, e.g., in Fig. 3, suppose the min-width for $V_i$s is violated, but its children $img_i$ and $p_i$ are not. Applying ($V_i$, *red-col*) does not change $V_i$'s width, only its children's width.

- Applying *red-col* (*red-row*) on $n$ is redundant if none of its descendants is violating min-width (max-width) constraints.

- Applying *red-margin* if (i) max-width is violated for $n$, i.e., width($n$) > max-width($n$) (*red-margin* increases the width of $n$), or, (ii) $n$ violates its min-width constraint and maximum margin reduction is not enough, i.e., width($n$) + max-margin-reduction($n$) < min-width($n$). (cf. *red-margin* fix).

The second optimization tries to detect *equivalent* fix-chains in a DTG which correspond to re-orderings of *independent* fixes.

**Independent Fixes.** Two fixes $f_1$ and $f_2$ on nodes $n_1$ and $n_2$, respectively, are said to be *independent* if (i) applying one fix does not disable the other, and (ii) the $T'$ obtained after applying both $f_1$ and $f_2$ does not depend on the order in which the fixes are applied, i.e., the fixes *commute*. The notion of independence allows us to explore only the *representative* fix-chains of enabled fixes. Most fixes in Table 2 commute with each other, leading to the same DT irrespective of the application order. However, depending on the given constraints, fixes may disable each other. Fig. 6 illustrates a portion of the DTS for a design with three boxes $B_1$, $B_2$ and $B_3$. In the initial state $q_1$, the *min-width* criteria for both $B_2$ and $B_3$ is violated, i.e., bad nodes are $\{B_1, B_2, B_3\}$. There are two ways to fix it: by applying fix $f_1$ (red-col) on $B_1$ or by applying fix $f_2$ (red-margin) on $B_2$ and $B_3$. The two fixes are dependent on each other: applying $f_1$ leads to $q_2$, with no bad nodes and hence $f_2$ is disabled. Similarly, applying $f_2$ at $q_1$ disables $f_1$ in $q_3$.

Fixes which only affect the subtree of the target node cannot disable each other because they apply to non-overlapping subtrees, e.g., in Fig. 4, fixes *red-margin*, *red-row* and *red-col* can be applied to each of the nodes $V_1$, $V_2$, $V_3$ and $V_4$ and are independent of each other. For example, the fix-pair ($V_2$, *red-margin*) enabled at state $q_1$, remains enabled at $q_4$ and $q_5$, after applying fixes to $V_1$. Efficient exploration of graphs with dependent transitions is a hard problem: several generic *partial-order* reduction methods [25, 22, 23] for program state space exploration have been developed. We extend these methods to our problem by exploiting the fact that states in a DTG correspond to trees.

**Pre-order Fix Chains.** A pre-order *linearization*, $pre(T)$ of a DT $T$ is a node sequence obtained by traversing $T$ in pre-order. Because DTs are ordered, a unique linearization, $\widehat{pre}(T)$, exists for each DT $T$. Given a fix chain $s = (n_1, f_1), \cdots, (n_k, f_k)$, we define the *node-projection* of $s$, $s\downarrow$, as the sequence $n'_1, n'_2, \cdots n'_l$ such that (i) $n_1 = n'_1$ and $n_k = n'_l$, (ii) $s\downarrow$ preserves the node order of $s$ and does not stutter, i.e., for all $1 \leq i < k$, $n'_i \neq n'_{i+1}$. A fix chain $s$ over $T$ is said to be *pre-order* iff $s\downarrow$ is a sub-sequence of $\widehat{pre}(T)$. For example, in Fig. 3, the fix-chains [($H_1$, red-margin), ($H_1$, red-col), ($V_1$, red-margin)] and [($V_1$,red-margin), ($V_2$,red-margin)] are pre-order but [($V_1$, red-margin), ($H_1$, red-col)] is not.

We observed that fixes on higher tree nodes affect a large number of nodes and few such fixes (in contrast to lower node fixes) are sufficient to yield most desired DTGs. Pre-order fix chains are biased towards such fixes; fixes are applied on the parent node first, followed by children. Also, pre-order fix chains process siblings in a fixed order, allowing us to avoid permutations of independent fixes and narrow the search space. We therefore constrain EXPLOREDTG to search only for pre-order fixes by checking *on-the-fly* if the current fix-chain is a sub-sequence of the DT linearization. Given a tree $T$ with $N$ bad nodes, number of sub-sequences of $pre(T)$ is $O(N \cdot p!)$ where $p = |F| + 1$. Although still large, this bound is significantly lower than the earlier bound $O(N! \cdot p^N)$.

**Ranking Fix Sequences.** DECOR ranks the obtained fix chains $\Lambda$ (Alg. 1) as follows. Shorter fix-chains are ranked higher than longer ones. For fix-chain $s$ and $s'$ of equal length, $s$ is ranked higher if between nodes $n_i$ and $n'_i$ at $i^{th}$ index, resp., $n_i$ has a higher tree height than $n'_i$. We can also shorten fix-chains to remove irrelevant fixes before ranking them (cf. Sec. 7).

## 7. IMPLEMENTATION AND EVALUATION

We implemented DECOR as a plugin to Maqetta [11], an open-source WYSIWYG editor, which allows creating *designs* by drag-drop and then specifying CSS properties. The plugin extracts the design as a JSON file and sends it to a backend server (node.js [13] application), which provides responsive design suggestions. The backend takes in user-specified constraints and breakpoints also in JSON format. We run a simple checker to identify any inconsistency in the input constraints before running DECOR, e.g., min-width is higher than the max-width constraint for an element.

DECOR can operate in two modes - *default* and *interactive*. In the *default* mode, DECOR is fully automated: it finds all possible fix chains for each breakpoint, ranks them, applies the highest ranked fix-chain to get the repaired design for that breakpoint and proceeds to the next smaller breakpoint. In the *interactive* mode, user shrinks the design up to a certain breakpoint, identifies constraint violations, adds new constraints and invokes DECOR to get recommendations. She can add/remove/update constraints and run DECOR again to get a different set of recommendations. Finally, the set of obtained design trees, one for each breakpoint, are encoded into a pair of HTML and CSS files using the technique in [33]: each tree is encoded into CSS rules separately and then combined using media-queries [12]. If multiple recommendations exist, each of them gets its own CSS file. DECOR is 15K lines of JavaScript code (around 6K lines for repair computation).

We evaluate DECOR from two different perspectives. **(Q1)** What are the characteristics of recommended fix-chains and how effective are the pruning strategies (cf. Sec. 6) ? **(Q2)** To test the overall usability of the tool, understand the benefits and drawbacks of the proposed approach. For the first part, we ran DECOR in default mode and thoroughly study the recommended fix chains and computation times. For the second part of evaluation we conduct a preliminary user study of the tool in the interactive mode.

**Benchmarks.** No standardized set of benchmarks for responsive designs exist. However, several online resources point to the popular responsive patterns and web-sites implementing them, e.g., Neil's slides [15] and the Marcotte's book [31] provide a comprehensive overview of responsive design patterns. From these references, we selected web-sites which implement the popular responsive patterns, e.g., *column-drop, column-flip, feature-items, feature-shuffle, mostly-fluid, mondrian, tiny-tweak, layout-shifter, gallery, top-nav* etc. [15] All these transformations can be obtained using the set of fixes implemented in DECOR. We created two sets of benchmarks: (a) *synthetic* designs with few boxes, each of which demonstrates one or two responsive behaviors, and (b) *real-world* designs. For the latter, we selected 7 web-sites (Table 3) having multiple responsive patterns and created mockups in Maqetta faithful to the original web-sites: we ignored some style annotations and few elements irrelevant to the responsive behavior. Table 4 shows the benchmarks (synthetic followed by real). In total we used 20 mockups for evaluating DECOR in the default mode. Drawing each initial design took about 1-2 hours. All mockups are drawn originally for desktop width (1400px).

**Defining Constraints.** For the first part of evaluation, we defined constraints on properties, e.g., width and margin, for each bad UI element. Most constraints relate to min/max width or margins of elements and apply to all breakpoints. Breakpoint-specific constraints are only a few (5%): those for *hiding* an element and fixing its font-size. To specify constraints for a breakpoint, we shrink the browser size to the breakpoint and obtain, say, the current width for a bad element or its parent. Next, we add constraints to allow DECOR to provide recommendations which mimic original design transitions. Constraint specification for synthetic designs to obtain desired behaviors was easy. However, constraints for the real designs to mimic original behaviors required a few iterations to get right. Specifying constraints for real benchmarks took 3mins (five-steps, portfolio, palantir) to 10mins (modernizr, overview).

| Mockup Name | URL |
|---|---|
| overview | http://foodsense.is |
| portfolio | http://www.bradsawicki.com |
| five-steps | http://www.fivesimplesteps.com/ |
| orestis | http://www.orestis.nl |
| modernizr | http://modernizr.com/ |
| palantir | http://palantir.net |
| trent-walton | http://trentwalton.com/ |

**Table 3: Sources of real responsive pages in benchmarks.**

## 7.1 Tool Performance

We implemented the following fixes in DECOR: *red-col, red-margin, red-row, hide-node, mod-nav, mod-text*, and *mod-line-height* (cf. Table 2). We conducted all experiments in default mode on a Mac Snow Leopard with 2.3 GHz Intel Core i5 processor and 4GB memory. To gain insight into the running times, fix-chain characteristics and effect of optimizations, we implemented three exploration modes in DECOR.

- Dynamic Depth limit (DDL): Once DECOR finds a fix-chain of length $l$, it restricts its search to all fix chains of depth $l$. This mode finds at least one fix-chain, if it exists, and then does not waste time searching for longer chains.
- Bounded exploration: Given a particular bound $l$, DECOR restricts its search to all fix-chains of length less than $l$ in DTG. Multiple bounds are evaluated. In contrast to DDL mode, a fix-chain may not be found if $l$ is too small.
- Lazy mode: Here, DECOR stops after finding single fix-chain (as opposed to earlier modes), which is minimized by greedily removing fixes (in no particular order) and checking if the remaining chain repairs the tree. This mode aims at finding a single, short fix-chain quickly.

The number of UI elements (boxes) vary between 7 to 109, and the tree height between 2 and 6, the number of constraints between 2 to 21 in the benchmarks. Table 4 shows the results on 20 benchmarks with the *DDL* mode for each of the 3 breakpoints. For each breakpoint, the number of bad nodes and fix-chains (recommendations) found, the lengths of fix-chains and the time is shown. Although the number of bad nodes (cf. Sec. 5) is large in many designs, a short fix-chain is sufficient, e.g., for *portfolio*, the number of bad nodes is 19 in stp breakpoint but requires a fix-chain only of length 2 to repair it. Note that it may be hard for the designer to figure this out manually: she may perform too many unnecessary fixes for the bad nodes before realizing that a shorter fix exists. For real benchmarks, most fix-chains have length $2-6$ except for stp, e.g., $11-13$ for *modernizr*. This is because the same fix *red-margin* is applied on 6 similar images in *modernizr*.

The ranking heuristics in DECOR perform well on most real designs in mimicking the original design transitions accurately. DECOR finds fix-chains that mimic the behavior of all real benchmarks among five top-ranked recommendations: for all benchmarks except *five-steps* (rank 2) and *portfolio* (rank 4), the top recommendation matched the real behavior. In *five-steps* and *portfolio*, DECOR finds a shorter fix-chain satisfying constraints than the one that captures the real behavior and ranks it higher.

Overall, DECOR successfully computes multiple possible fix-chains for multiple breakpoints on realistic mockups with 100 elements, within 30 seconds. Thus, it can be used interactively to perform multiple design updates. DECOR owes its performance to a series of optimizations discussed earlier (Sec. 6). Across all benchmarks, the maximum memory used by DECOR is $<$ 100 MB.

Table 4 shows that with a small set of constraints we are able to infer realistic fix-chains. For benchmarks *portfolio*, *modernizr*, *palantir*, the number of constraints required is larger (>10); however, this is because of multiple similar elements: the *unique* constraints here are 5, 10, and 7 respectively.

**Effect of Optimizations.** We show the effect of optimizations (cf. Sec. 6) on a selected set of 4 real benchmarks (other benchmarks

| | | | Tablet (width 1024px) | | | | Small tablet portrait (stp, width 600px) | | | | Mobile portrait (mop, width 300px) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mockup | # Nodes / Height | # Constraints/ # C-nodes | # Bad node | # Fix Chain | Chain Len. Min/ Max/ Avg | Time | # Bad node | # Fix chain | Chain len. Min/Max/Avg | Time | # Bad node | # Fix chain | Chain len. Min/Max/Avg | Time | Total Time |
| mostly-fluid | 12/3 | 3/3 | 1 | 1 | 1/1/1 | 0.105 | 0 | ✓ | N/A | 0.033 | 8 | 2 | 1/2/1.5 | 0.206 | 0.353 |
| column-drop | 11/3 | 2/2 | 1 | 1 | 1/1/1 | 0.092 | 4 | 2 | 1/2/1.5 | 0.201 | 4 | 2 | 1/2/1.5 | 0.187 | 0.488 |
| tiny-tweak | 11/3 | 3/3 | 1 | 1 | 1/1/1 | 0.09 | 4 | 1 | 1/1/1 | 0.093 | 6 | 2 | 1/2/1.5 | 0.216 | 0.406 |
| mondrian | 7/2 | 7/4 | 4 | 1 | 1/1/1 | 0.146 | 6 | 2 | 2/3/2.5 | 0.188 | 7 | 4 | 2/4/3 | 0.197 | 0.562 |
| gallery | 11/3 | 6/6 | 1 | 1 | 1/1/1 | 0.098 | 8 | 2 | 1/2/1.5 | 0.146 | 8 | 2 | 1/2/1.5 | 0.165 | 0.416 |
| featured-items | 15/3 | 4/4 | 0 | ✓ | N/A | 0.078 | 10 | 4 | 1/3/2 | 0.323 | 10 | 4 | 1/3/2 | 0.475 | 0.888 |
| top-nav | 19/3 | 5/5 | 2 | 4 | 1/2/1.5 | 0.279 | 10 | 4 | 1/3/2 | 0.482 | 10 | 4 | 1/3/2 | 0.713 | 1.483 |
| nav-bar | 19/3 | 5/5 | 2 | 4 | 1/2/1.5 | 0.277 | 10 | 4 | 1/3/2 | 0.464 | 11 | 7 | 2/4/3.29 | 1.177 | 1.926 |
| hide-3 | 8/2 | 6/4 | 4 | 1 | 1/1/1 | 0.112 | 5 | 2 | 3/4/3.5 | 0.14 | 7 | 4 | 2/4/3 | 0.215 | 0.475 |
| hide-p | 11/3 | 2/2 | 1 | 1 | 1/1/1 | 0.103 | 7 | 1 | 1/1/1 | 0.059 | 6 | 2 | 1/2/1.5 | 0.173 | 0.345 |
| hide-h | 11/3 | 3/3 | 2 | 1 | 1/1/1 | 0.113 | 7 | 2 | 2/3/2.5 | 0.216 | 2 | 1 | 1/1/1 | 0.073 | 0.412 |
| hide-n-font | 11/3 | 3/3 | 1 | 1 | 1/1/1 | 0.09 | 4 | 1 | 1/1/1 | 0.091 | 0 | ✓ | N/A | 0.035 | 0.223 |
| rpe-mockup | 35/5 | 4/4 | 5 | 1 | 1/1/1 | 0.166 | 5 | 4 | 2/4/3 | 0.558 | 0 | ✓ | N/A | 0.06 | 0.794 |
| overview | 69/5 | 9/8 | 6 | 2 | 4/5/4.5 | 3.462 | 10 | 5 | 3/5/4.2 | 5.928 | 11 | 4 | 3/5/4 | 12.402 | 21.888 |
| portfolio | 31/3 | 16/15 | 4 | 1 | 1/1/1 | 0.351 | 19 | 2 | 2/3/2.5 | 0.847 | 17 | 2 | 2/3/2.5 | 0.64 | 1.845 |
| five-steps | 95/6 | 8/8 | 12 | 6 | 2/3/2.5 | 5.926 | 12 | 24 | 3/7/4.83 | 17.737 | 5 | 4 | 2/4/3 | 4.067 | 27.738 |
| orestis | 109/6 | 10/10 | 1 | 1 | 1/1/1 | 1.021 | 9 | 16 | 5/9/7 | 22.389 | 10 | 1 | 4/4/4 | 1.841 | 25.258 |
| modernizr | 62/5 | 21/16 | 8 | 8 | 3/5/4 | 5.037 | 16 | 4 | 11/13/12 | 7.748 | 19 | 2 | 6/7/6.5 | 3.193 | 15.986 |
| palantir | 62/4 | 11/8 | 3 | 2 | 2/3/2.5 | 0.964 | 11 | 12 | 4/8/6 | 12.975 | 10 | 4 | 5/7/6 | 2.478 | 16.428 |
| trent-walton | 27/4 | 5/4 | 0 | ✓ | N/A | 0.09 | 16 | 1 | 12/12/12 | 0.979 | 12 | 1 | 8/8/8 | 1.254 | 2.33 |

**Table 4: Fix data for all benchmarks in DDL mode with all optimizations applied. All times in seconds. Column #C-nodes denotes the number of constrained nodes. ✓denotes that the design satisfies constraints and no fix is needed.**

show similar results) in Fig. 7(a). *Basic* column corresponds to applying hide-node, mod-text fixes initially followed by full DTG exploration. *Basic + preorder* corresponds to searching only for pre-order fix chains (Sec. 6). *Basic+rfp* stands for combining basic mode with redundant fix pruning (rfp). *Basic+ preorder + rfp* stands for combining all above optimizations, but allowing permutations of fixes on same node. *All* column combines all optimization heuristics, i.e., basic, preorder fix-chains, pruning redundant fixes and eliminating fix permutations on a node. The results show that preorder fix-chains and *rfp* provide the most significant improvement in run-times. For small benchmarks, the effect of optimizations is not significant but they are essential for real benchmarks: running *overview* example with only basic optimization applied took more than 3 hours for one breakpoint (stp), whereas it came down to almost 30s after applying all optimizations.

Figure 7(b) shows the correlation of enabled fixes and run-time for *overview* benchmark. We observe that the runtime of DECOR increases linearly with the number of fixes enabled; optimizations are able to reduce this value, thus leading to improved performance. Also observe that preorder fix-chains have a drastic effect in reducing the number of enabled fixes, from tens of thousands to less than 500; rfp brings it down further. Finally, combining all optimizations reduces enabled actions to only 128.

**Lazy mode.** Fig. 7(c) compares the runtime of DDL (find all fix-chains) and Lazy (find one fix-chain) modes, with all other pruning optimizations. In most cases, the top-ranked fix-chain in DDL and the one found by the Lazy mode are of equal lengths. In general, the Lazy mode returns the first fix much sooner than the DDL mode. However, for some examples the Lazy mode takes longer: if a fix-chain is long and has no redundant fix, the minimization time in Lazy mode outweighs the benefit of searching for only one fix chain. Although DDL does not minimize fix-chains explicitly, it explores multiple fix-chains and hence can find short chains also.

**Limitations.** The current version of DECOR requires users to specify constraints on individual nodes, not groups. DECOR cannot extract design trees from existing designs currently. We therefore specify original designs in Maqetta [11] WYSIWYG editor, which is time-consuming and laborious. We plan to handle legacy designs in future. Better support for visualizing the interaction of constraints with fixes is needed for a better debugging experience. Finally, DECOR is restricted to its fix library and chosen responsive patterns during recommendation. While we support the common fix actions, others are not implemented yet, in particular, those for changing order of elements and navigation patterns which require complex JavaScript interaction.

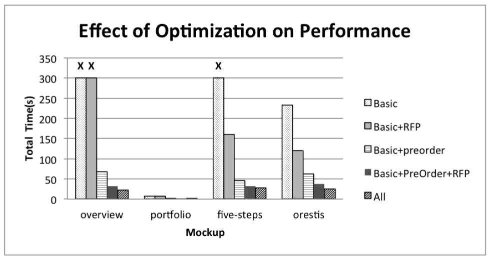
## 7.2 Preliminary User Study

We conducted a preliminary study to gauge the manual effort in DECOR and compare with Adobe Edge Reflow (AER) [2], a commercial tool, which enables creating designs for multiple breakpoints by direct manipulation. We created a web UI to specify constraints (in JSON format), run DECOR on various benchmarks and view recommendations. We chose 3 participants (1 female, mean age=26.33, sd=2.52) with minimal expertise in UI programming, i.e., who are familiar with HTML/CSS based design using WYSIWYG editors but are unfamiliar with responsive design.

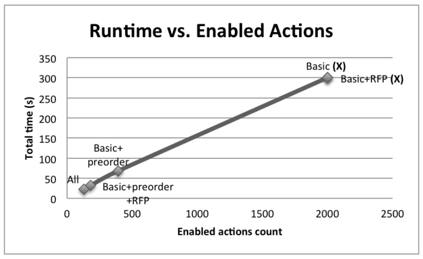
The study was conducted in two phases. First, the participants were given a tutorial on both AER (using web site [2] tutorials) and DECOR (drawing mockups in Maqetta with direct manipulation, writing constraints in JSON format, and using browser tools to inspect output design features). This was followed by a walk-through of DECOR on the *overview* benchmark: drawing mockup, setting constraints and viewing recommendations. The effect of various fixes and interaction between constraints and fixes was explained. The participants could modify the constraints iteratively via the UI, run DECOR and inspect changes in recommended designs.

In the second phase, we asked the participants to create responsive UIs from an initial design for 3 breakpoints using both AER and DECOR, in their preferred order. We chose 3 UIs: *nav-bar* (simple), *portfolio* (slightly complex) and *modernizr* (complex). Here, complexity is measured using the number of UI elements, responsive transitions and the number of constraints. At the end of the task, the participants were asked to answer an evaluation questionnaire set. The questions included rating the difficulty in adding constraints and comparison of design effort with DECOR as opposed to AER. To complete the tasks, the participants take about 10 minutes with DECOR and about 20 minutes with AER. The user feedback suggested the following benefits of DECOR over AER.

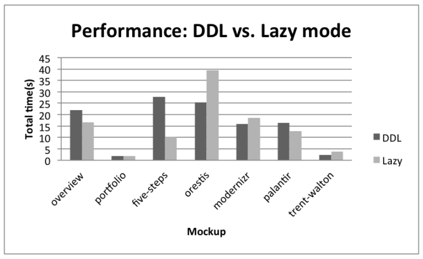
**Repeated manual effort.** In AER, the participants had to manually rearrange UI elements for each breakpoint. In contrast, constraints on elements in DECOR trigger responsive behavior directly. For



(a)



(b)



(c)

**Figure 7: (a) Effect of optimizations on runtime. (b) Effect of number of enabled actions on runtime for *overview* benchmark (c) Runtime for benchmarks in DDL and lazy modes (all optimizations enabled). Column 'X' denotes a timeout (5min).**

example, in the *modernizr* benchmark, the team images are arranged in 2x3 grid initially. Each time a user tries to modify the grid layout in AER for lower breakpoints, say to 3x2 or 6x1, she has to resize and rearrange each of the images. On the other hand in DECOR, the same constraint, e.g., min-width of an image, triggers layout grid changes for all the breakpoints. Similarly, in *portfolio*, the grid contains 12 elements and for each breakpoint, all of them either have to be relocated, resized or the corresponding CSS property updated carefully in AER. In DECOR, the constraints for one element in the grid can be quickly copied to other siblings. Moreover, specifying constraints on only a subset of elements is often sufficient to obtain the desired responsive behavior.

**Useful Recommendations.** In contrast to DECOR, AER provides no recommendations. The designer bears the onus of choosing the right responsive patterns in AER, requiring deep knowledge of responsive designs, e.g., in the *nav-bar* benchmark (arranged as a single row grid in the *tablet* breakpoint), DECOR gave two recommendations for *mop*: replace it by *select*, or reduce the number of columns and place the navigation anchors in different rows. One of the participants did not think of the *select* option with AER and found the suggestion by DECOR very helpful.

**Avoid Trial and Error.** Lack of recommendations in AER forced the participants to try out and undo different design changes one-by-one, which was time-consuming. For *nav-bar*, a participant tried to rearrange the elements in top *navbar* in different ways, in a single column or multiple rows and finally replaced them with a *select* menu. In contrast, in DECOR, she specified only the min-width constraint and obtained all the three recommendations within a minute. Participant answered that even if each constraint corresponds to a single fix, it is easier in many cases to specify constraints than guessing the correct element to fix manually.

Participants remarked that AER's superior direct manipulation features (compared to Maqetta) allowed them to create initial designs faster. We also received multiple feature requests for DECOR: ability to specify non-numerical constraints (too narrow or wide), constraints on multiple elements simultaneously, tool recommendations to hide elements. We plan to incorporate these requests in the tool and conduct a user study over a bigger participant set in future. Although thinking in terms of constraints required more work for simple designs, participants were quite enthusiastic about using DECOR as the design complexity increased. In summary, the participants found DECOR's approach of using constraints and providing design recommendations more appealing than direct manipulation based design in AER.

## 8. RELATED WORK

Constraints are widely used for *macro*-typography (see [27, 35] for a detailed survey) to specify UI object sizes and relative alignment [21, 18, 24]. Constraints are central to iOS (*Auto Layouts*) [10], which also allows visual constraints. However, these systems assume a dedicated layout engine: the solver computes the exact pixel locations of elements for a particular viewport, which is used on-the-fly by the renderer. Further (cf. Sec. 1) these systems search over a narrow design space: fixes proposed by the solver can re-size (shrink, expand) boxes but not re-arrange them. Annotation-based libraries, e.g., Bootstrap [7], Zurb foundation [17], have a steep learning curve and require deep HTML/CSS expertise [30, 33] to map arbitrary designs to the framework-specific annotations.

Recent work investigates specifying constraints by direct manipulation [35, 26]. None of these tools provide multiple design recommendations: [35] provides conflict explanations, but each explanation may correspond to multiple fixes leading to several possible designs. Remorph [20] retargets existing web pages by *shrinking* element widths based on viewport without user-specified constraints: this may result in *too-wide* elements and unaesthetic designs. Bricolage [29] is a machine learning based approach to retargeting based on a web-based corpus of designs.

In contrast, we perform constraint-based design tree repair and provide a small set of recommendations via a set of novel design-space pruning and ranking techniques. We avoid a fully symbolic encoding, as the latter requires logical axiomatization of trees (with quantifiers), making implementing optimizations and browser-based deployment much more complex. Encoding the solution DTs as a HTML/CSS file [33] along with media-queries allows us to employ the off-the-shelf, heavily optimized rendering engines of browsers to parse and render HTML/CSS for responsive behavior. Adaptive document layout techniques were presented in [32] based on solving constraints over user-specified layout templates and a dedicated rendering engine. In contrast, we target HTML/CSS designs based on nested box layouts (designer only specifies constraints on elements not templates) and exploit off-the-shelf renderers. Another option is to use JavaScript for on-the-fly layout computation in browser; however, this overloads the mobile devices unnecessarily and also fails to exploit the native optimized renderer.

## 9. CONCLUSIONS

We presented a systematic approach for inferring multi-device responsive designs from a given UI design and user-specified constraints. Our tool DECOR assists the designer by providing design recommendations interactively. DECOR aims at improving the productivity of experienced designers for creating responsive designs as well as reducing the learning curve for HTML/CSS and responsive designs for unskilled designers. DECOR is able to provide realistic list of recommendations in a short time for real-life designs. We plan to add constraints for multiple elements, e.g., by direct manipulation, and improve debugging experience. Other planned features include adding *inverse* fixes from lower to higher breakpoints, and fixes which adapt image resolution to devices.

## References

[1] Adobe Dreamweaver CS6. `http://www.adobe.com/products/dreamweaver.html`.

[2] Adobe edge reflow cc. `http://html.adobe.com/edge/reflow/`.

[3] Balsamiq. `http://www.balsamiq.com/`.

[4] Cascading Style Sheets Level 2 Revision 1 (css 2.1) Specification. `http://www.w3.org/TR/CSS21/`.

[5] CSS flexible box layout module. `http://www.w3.org/TR/css3-flexbox/`.

[6] Food sense website. `http://foodsense.is/`.

[7] Get bootstrap. `http://getbootstrap.com/css/`.

[8] Html5. `http://www.w3.org/TR/html5/`.

[9] Interactive wireframe software and mockup tool. `http://www.axure.com/`.

[10] iOS Auto Layouts. `https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/Introduction/Introduction.html`.

[11] Maqetta. `http://maqetta.org/`.

[12] Media queries, Mozilla MDN. `https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Media_queries`.

[13] node.js. `http://nodejs.org/`.

[14] Responsive designs in a snap (full version). `http://researcher.watson.ibm.com/researcher/files/in-nishant.sinha/responsive.pdf`.

[15] Theresa Neil: Mobile design strategic solution. `http://www.slideshare.net/theresaneil/mobile-design-strategic-solutions/`.

[16] Why 2013 is the year of responsive web design. `http://mashable.com/2012/12/11/responsive-web-design/`.

[17] Zurb foundation. `http://foundation.zurb.com/`.

[18] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, December 2001.

[19] Greg J. Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint cascading style sheets for the web. UIST '99, pages 73–82, 1999.

[20] G. L. Bernstein and S. Klemmer. Towards responsive retargeting of existing websites. UIST'14 Adjunct.

[21] A. Borning, B. N. Freeman-Benson, and M. Wilson. Constraint hierarchies. In *Over-Constrained Systems*, pages 23–62, 1995.

[22] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press.

[23] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[24] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock. Automatically generating personalized user interfaces with supple. *Artif. Intell.*, 174(12-13):910–950, 2010.

[25] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Secaucus, NJ, USA, 1996.

[26] T. Hottelier, R. Bodik, and K. Ryokai. Programming by manipulation for layout. In *UIST*, 2014.

[27] N. Hurst, W. Li, and K. Marriott. Review of automatic document formatting. In *Proceedings of the 9th ACM symposium on Document engineering*, DocEng '09, pages 99–108, 2009.

[28] P. M. Marden Jr. and E. V. Munson. Today's style sheet standards: The great vision blinded. *IEEE Computer*, 32(11):123–125, 1999.

[29] R. Kumar, J. O. Talton, S. Ahmad, and S. R. Klemmer. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2197–2206, 2011.

[30] H. W. Lie. Cascading style sheets. PhD thesis, University of Oslo, February 2006.

[31] E. Marcotte. *Responsive Web Design*. A Book Apart, 2011.

[32] E. Schrier, M. Dontcheva, C. Jacobs, G. Wade, and D. Salesin. Adaptive layout for dynamically aggregated documents. In *IUI*, pages 99–108, 2008.

[33] N. Sinha and R. Karim. Compiling mockups to flexible uis. In *ESEC-FSE*, 2013.

[34] L. Wroblewski. *Mobile First*. A Book Apart, 2011.

[35] Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. The auckland layout editor: An improved gui layout specification process. UIST '13, 2013.