Compiling Mockups to Flexible UIs

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

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

ABSTRACT

As the web becomes ubiquitous, developers are obliged to develop web applications for a variety of desktop and mobile platforms. Re-designing the user interface for every such platform is clearly cumbersome. We propose a new framework based on model-based compilation to assist the designer in solving this problem. Starting from an under-specified *visual* design *mockup* drawn by the designer, we show how faithful and flexible web pages can be obtained with virtually no manual effort. Our framework, in sharp contrast to existing web design tools, overcomes the tough challenges involved in mockup compilation by (a) employing combinatorial search to infer hierarchical layouts and (b) mechanizing adhoc principles for CSS design into a modular, extensible rule-based architecture. We believe ours is the first disciplined effort to solve the problem and will inspire rapid, low-effort web design.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design

Keywords

Layout inference, Mockup-based design, HTML, CSS

1. INTRODUCTION

Web user interface (UI) design is a fundamental part of web application design [42, 43]. The UI quality often directly influences the adoption of the application and therefore designers spend significant effort in getting the user interface right. An important component of UI design is *static* design, involving selection of elements, their layout and styling (coloring, typography) for a particular web page view. Static web design requires a deep understanding of interaction between (a) page structure definition in HTML [8] and (b) the layout/style modeling in CSS [3, 5, 37]. Both of these are low-level technologies and CSS, in particular, has a steep learning curve. This makes HTML/CSS authoring extremely labor-intensive and time-consuming [32, 27, 30]. In particular, CSS styling largely remains an art [3, 37, 30]: CSS does not include constructs for specifying complex layouts directly; designers must carefully work around various CSS features manually to obtain desirable layouts for different platforms. Consequently, web design often involves two people working closely [43, 28]: a web designer, who uses a graphic editor, e.g., Adobe Photoshop, to design the look in a drag-drop fashion, and a developer, who encodes the design based on her HTML/CSS expertise. This partnership is often unproductive due to the strong coupling and feedback delays: minor changes by the either party may cause significant grievances for the other.

With the web becoming ubiquitous [4] on mobile devices with multiple form factors, the problem has become worse. Applications must be designed for multiple devices simultaneously and adapt to rapidly evolving technology besides having a shorter time-to-market. Moreover, we observe an increasing involvement of non-expert developers in the new echo-system, who are barely skilled in HTML/CSS. The prevalent partnership model hardly scales to the increased development demand. In fact, we now need tools which shield the developers from the low-level technologies and minimize distractions, thus enabling low-effort, rapid web development.

Mockup-based design [42, 43] offers an appealing solution to rectify the current dismal state of affairs. A *mockup* (also called a *wireframe*) is a rough visual depiction of the desired design, drawn in a software tool (a WYSIWYG editor) and reflects the designer's intention visually. A designer can draw mockups easily in an editor by using drag-drop operations on UI widgets from a palette collection and apply menu-based style selections. Mockups are essentially technology-independent: a designer only needs a superficial understanding of the underlying technology to draw mockups. Further, a single mockup may be used to obtain different implementations for multiple devices via device-specific compilation. Finally, these designs are reusable even as the web technologies evolve. Clearly, mockup-based design is advantageous in multiple ways.

To bring this vision to reality, several WYSIWYG editors have been designed [28], e.g., Adobe Dreamweaver [1], Microsoft Expression Web [12], over the past decade. Although these editors assist the designer with content authoring, it is widely accepted that they have failed to inspire ubiquitous mockup-based development. The key hurdle is that enormous, low-level manual effort and a deep knowledge of HTML/CSS is required to work with these editors. Moreover, the tool requires the designer to not only specify the UI elements constituting the core design but also the precise layout for those elements. This often leads to exposing the designer to underlying low-level details of HTML/CSS, which was originally intended to be kept under-the-hood. Moreover, in return for the extreme manual effort, most editors generate fixed designs which cannot adapt to browser *viewport* or device changes. Thus these tools generate lot of user dissatisfaction and are deserted by unskilled designers. More recent WYSIWYG tools restrict themselves to quick design prototyping [2, 9, 11] and do not promise high-quality code translations. This brings up the natural question:

Why can't they do better? How hard is compiling mockups?

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
ACM 978-1-4503-2237-9/13/08
http://dx.doi.org/10.1145/2491411.2491427

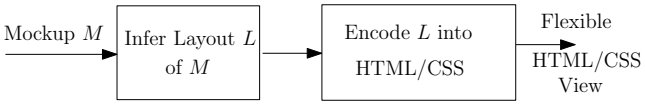

Figure 1: Overview of the two-phase approach for encoding mockups to flexible web pages.

Automatic high-quality compilation of mockups is indeed quite non-trivial. There are two main challenges:

Challenge 1 (Infer Layout) A mockup $\mathcal{M}$ is a freely-drawn, under-specified design schema over UI elements, where the page layout is not explicit. A tool must mine and generalize from the latent hierarchical structure in $\mathcal{M}$ to extract the *right* layout for $\mathcal{M}$, i.e., the local vertical/horizontal flow of the page content, which also preserves the relative sizes and alignment of individual elements. This involves solving a non-trivial combinatorial search problem over the set of possible layouts.

Challenge 2 (Encode in HTML/CSS) The second problem is how to encode the inferred layout in an HTML page faithfully. This requires a deep understanding of CSS semantics, which is complex and has no support for encoding layouts directly. Moreover, to generate superior encodings, the tool must incorporate the poorly documented CSS design principles used by designers in practice.

Somewhat surprisingly, there exists no systematic framework to address the above two challenges for web design. Researchers have employed constraint solvers [29, 38, 22] to address the first challenge of finding layouts and personalized interfaces [25, 24] in context of standalone desktop-based applications. These efforts assume that a dedicated rendering engine is used, which can readily interpret the pixel value solutions provided by the solver. The web, unfortunately, fails to meet this requirement.

Each off-the-shelf web browser embeds a complex rendering engine [18, 7, 40], optimized over many years, which can primarily interpret HTML/CSS/JavaScript commands. Modifying the layout engine is therefore not advisable; instead, we would like to exploit the existing engines maximally. Using JavaScript for layout is undesirable, specifically for mobile devices. The problem therefore becomes that of communicating layout information to the browser efficiently in terms of HTML/CSS *boxes*. This, in turn, requires us to express and search for layout in terms of a hierarchy of rectangular boxes, and infer a hierarchy of *container boxes* for UI objects, as opposed to pixel values. Box-based layouts have been investigated for text documents, e.g., LaTeX [34], and grids [29, 38], both of which are unsuitable (cf. Sec. 3) for laying out variable-sized boxes with heterogeneous content as in a web application mockup.

Similarly, most WYSIWYG editors dodge the second challenge and adopt the path of least resistance. This is not surprising because encoding CSS rules automatically is quite challenging (cf. Sec. 5). They use simplified encoding techniques, e.g., based on widely criticized HTML tables [16], and often resort to a *fixed* layout, which necessitates scrollbars and does not adapt to viewport or font resizes. The low-quality encodings generated by these tools, however, diminish the user experience and adoption significantly [28].

In this paper, we present a *systematic framework* to obtain faithful HTML/CSS designs from a mockup $\mathcal{M}$ automatically. Our encoding technique consists of two phases shown in Fig. 1. We first propose a new algorithm to infer an hierarchical layout L from $\mathcal{M}$ (Challenge 1) by recursively partitioning it into boxes which flow left-to-right (HBoxes) or top-to-down (VBoxes). This recursive partitioning extracts the implicit hierarchical structure of $\mathcal{M}$, besides capturing the relative size and alignment of elements. To address Challenge 2, we present a systematic modular architecture for encoding these layouts into HTML/CSS: each module consists of a set of mathematical rules for step-wise encoding L into HTML/CSS. The proposed architecture is extensible and allows easy fine-tuning of encoding rules. Further, along the lines of the original vision, we show how to construct flexible designs from a single mockup, which can adapt to multiple devices gracefully.

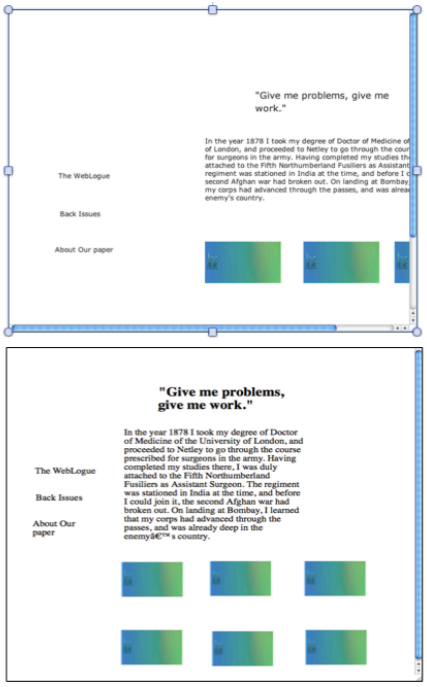

Figure 2: Comparison of a fixed layout (top) and a fluid-elastic layout (bottom) generated by our tool.

Fig. 2 shows the advantage of our flexible encoding over a fixed encoding: note how the fixed layout (on the top) gets cropped as the browser window re-sizes but the fluid layout (bottom) scales down to the smaller viewport size, with an *elastic* text formatting. We evaluate the framework on mockup benchmarks which model several common web UI design patterns and demonstrate high quality encodings of the original mockup in HTML/CSS automatically. We show that in absence of our method, generating these designs would have required significant manual effort.

Our main *contributions* are as follows. We present a systematic, automatic framework to encode mockups into faithful HTML/CSS designs with flexible layouts. The generated design may be viewed in an off-the-shelf browser without modifying the browser or changing the prevalent web markup echosystem: we exploit the browser layout engine to the maximum so as to achieve high-quality encodings. Our framework consists of a new technique to infer hierarchical layouts from mockups based on combinatorial search, together with a systematic, *extensible* set of rules to encode these layouts. Extensibility enables fine-tuning designs as well as adapting encoding to technology evolution. Finally, our tool enables non-experts to develop complex web page designs with low manual effort without exposing them to much of under-the-hood HTML/CSS technologies. To tackle difficult design scenarios, we enable the designer to interact with and guide the tool towards desired encodings. We evaluated our method on a set of mockups modeling common web design patterns. Our results shows that automatic, high-quality, flexible mockup encoding is indeed feasible using our tool, with hardly any user intervention.

The paper is structured as follows. Sec. 2 overviews HTML and CSS semantics and Sec. 3 defines mockups and layouts formally. Sec. 4 presents the hierarchical layout inference algorithm and Sec. 5 presents the modular encoding rules into CSS. Sec. 6 discusses our tool infrastructure and evaluation results. Sec. 7 presents the related work and Sec. 8 concludes.

2. A BRIEF TOUR OF HTML AND CSS

HTML consists of a set of markup conventions to describe the structure of a web document in a device-independent manner using a set of *tags*, e.g., *title*, headings *h1*, *h2*, etc., division *div*, paragraphs *p*, lists *ul*, anchors *a*, inputs *input*, forms, and so on. The content of the document is embedded in the HTML structure. The upcoming HTML5 [8] standard also includes tags for specifying multimedia content, e.g., *audio*, *video*, etc. .

A *CSS* style sheet consists of a set of *rules*: each rule r is composed of one or more *selectors* and a declaration. The declaration consists of a set of property-value pairs. Properties and their values determine the visual appearance, e.g., height, width, font, color, margin, padding, of the elements in the browser. Selectors determine the HTML elements to which the properties apply to. They are defined using relations between HTML tags, e.g., *div* > *p* selects all paragraph elements which are children of some *div* element. The mutual alignment of the HTML elements, i.e., the *layout*, is also specified using CSS (discussed later). The CSS standards [3, 5] specify how properties may be inherited among the elements in a page's HTML structure and how rules are prioritized depending on their order or specificity [3]. We will primarily focus on CSS2.1 [3] in this paper because it is implemented by browsers to a reasonably complete extent and is well-known among web designers.

Structure/Style Separation. CSS rules may be specified inline with the HTML specification, but such mixing of style and structure is discouraged [3, 48]. To enhance modularity, CSS and HTML should be specified separately. The coupling between CSS and HTML is achieved by annotating HTML elements with label(s) corresponding to CSS style object(s) which may affect their appearance (see example below). This separation between HTML/CSS has multiple advantages, e.g., the order of specification of blocks in HTML remains independent of their positioning in the actual page layout. Moreover, blocks may be moved around in the hierarchy arbitrarily without changing their style.

Box Model Semantics. The semantic structure of a web page consists of a set of boxes: a box may be simple or compound (contain other boxes). Boxes which start on a new line and have a *line-break* after them are called *block* boxes (blocks, in short); otherwise, they are called *inline* boxes. Block boxes take 100% width of their parent unless specified; inline elements, e.g., *span*, only expand to contain the content inside them. The boxes are specified using specific HTML tags, e.g., the division (*div*) HTML tag is commonly used to specify a block box. The position and size of each box is given by a set of attributes: top, left, height and width. Boxes also have other style properties, e.g., border, padding, margin, color, font, etc. .

Margin, Padding, Border of Boxes. The margin property of a box B specifies the distance of an adjacent box from the border of the B. The padding property specifies the distance between the border and the box contents. Both margin and padding may be specified along four directions, i.e., left, right, top and bottom. These attributes are conveniently visualized using a *3-box figure* shown in Fig. 3. In CSS2.1, the size (height/width) of a box refers to the actual size of the content box (not the border box). Thus, the combination of height, width, border, margin and padding for boxes must be computed carefully during page design.

Example. Fig. 4 shows a HTML/CSS design. The HTML (top) consists of a *div* element is labeled with the *identifier* "content" and contains a header element (*h1*), a paragraph element (*p*), and two *div* elements containing text "box 1" and "box 2" respectively, both of which are labeled with *class mybox*. The visual attributes of boxes with labels *content* and *mybox* are specified in CSS separately on right. The CSS rules say that the top-level *div* element, referred to by its CSS selector *#content*, has a *padding* of *10px* on all sides and margin of *0px* on top and bottom, and of *100px* on left and right. The class *mybox*, denoted by selector *.mybox*, has two properties, height and width, both of value *100px*. The *mybox* class applies to both "box 1" and "box 2" *div*s, making

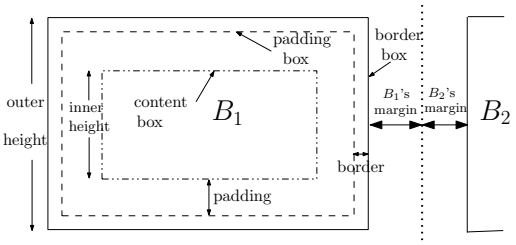

Figure 3: Anatomy of a CSS Block Box.

```
<div id="content">
 <h1>Main Content</h1>
 <p>Lorem ipsum etc..</p>
 <div class="mybox">
   box 1
 </div>
 <div class="mybox">
   box 2
 </div>
</div>
```

```
#content {
  padding: 10px;
  margin: 0 100px;
}

.mybox {
  height: 100px;
  width: 100px;
}
```

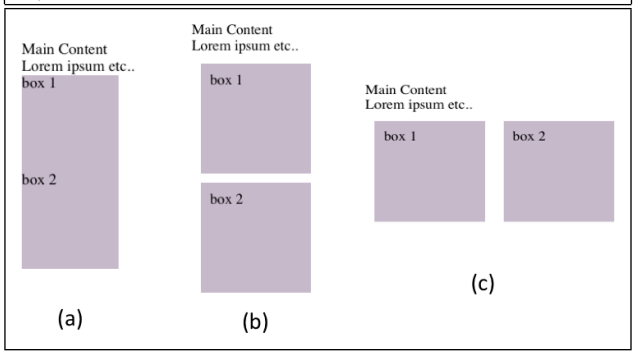

Figure 4: Interaction of CSS styling with HTML.

them of height/width 100px each. Fig. 4(a) shows how the above HTML/CSS is rendered in the browser. All elements are laid out vertically in the normal flow. Because there is no spacing between "box 1" and "box 2", they appear as a continuous block.

To improve the appearance of the design, we need to add margin and padding for both "box 1" and "box 2". We do this by adding *margin: 10px* and *padding: 10px* to the *.mybox* specification, which results in Fig. 4(b). Note how adding margins moves both the boxes slightly south-east and separates them from each other. Also, padding separates the text inside each *.mybox* from the box borders. Now, suppose that we want the boxes to be arranged left-to-right instead of top-to-down. For this, we add *float: left* to the *.mybox* specification (explained later). Fig. 4(c) shows the result. The final *.mybox* specification is

.mybox { height: 100px; width: 100px; margin: 10px; padding: 10px; float: left;}

Note how the separation between structure (HTML) and style (CSS) is beneficial: it enables us to obtain three different designs by modifying only the CSS without changing the HTML structure. Also, note that developing a simple web page where elements flow both vertically and horizontally requires a somewhat deep understanding of CSS box semantics, e.g., floats. The involved semantics have led, in practice, to multiple ambiguous interpretations of their behavior. Thus, modern CSS design is virtually an art form.

3. PRELIMINARIES: MOCKUPS, LAYOUTS

A mockup is a schema of the desired UI, based on representative UI elements [2, 11], drawn in a WYSIWYG editor [1, 12, 2, 11] using direct manipulation.

Mockups. Formally, a mockup $\mathcal{M}$ is a collection of rectangular

objects (*boxes*) m, each box having its visual properties, e.g., size, border, color, etc. . An object m may correspond to an HTML5 [8] UI element, e.g., *div, p, ul, label, input, img*, or a widget from a UI-library, e.g., [17]. Non-rectangular UI objects, e.g., text, images, circular objects, are represented by their bounding boxes in $\mathcal{M}$. Let the set $\mathcal{A}$ contain the common visual properties of boxes, e.g., *height, width, top, left* for their size/position, *font-size, border, background-color* for their look, as well as other CSS attributes. Also, the values for elements of set $\mathcal{A}$ are stored in the set $\mathcal{D}_{\mathcal{A}}$. Quantitative property values in $\mathcal{M}$ are specified in *pixels* or *ems* [3]. Formally, each object m is a tuple $r = (id, \sigma)$, where id is a string identifier and $\sigma : \mathcal{A} \rightarrow \mathcal{D}_{\mathcal{A}}$ is a map from attribute names to their values. Instead of writing $\sigma(height) = 10$px for m, we simply write m.height = 10.

Intuitively, a mockup corresponds to a free-hand drawing of UI elements on a canvas, e.g., that of a WYSIWYG editor, where the page layout is not explicit. Given such a drawing D, our approach extracts the absolute height, width and position attributes of the bounding boxes in the editor, along with their style properties, to obtain a mockup $\mathcal{M}$. Textual content is extracted out from the D and added to $\mathcal{M}$. Designer-specified container elements in the drawing are captured as parent-child relationships between boxes in $\mathcal{M}$. The layout inference algorithm then uses $\mathcal{M}$ to *guess* the layout intended by the designer.

Taxonomy of layouts. A layout refers to a particular arrangement of UI elements on a page. Web designers use an informal nomenclature to refer to common layout patterns. A *fixed* layout uses a fixed unit of measurement (pixels, typically) for positions and sizes of elements, independent of the browser viewport or the font size. Thus, fixed layouts are completely *static*, and do not change across font, resolution and viewport changes. While fixed layouts offer maximum consistency across multiple browsers, they are considered less user-friendly, e.g., they may require scrollbars on smaller viewports. A *relative* layout uses a relative unit of measurement, e.g., a percentage of the viewport size; the page content resizes to adapt to viewport or font size changes. Two common relative layouts are used: a *liquid* or a *fluid* layout and an *elastic* layout. Fluid layouts are characterized by specifying size of children as percentage of their parent sizes, while elastic layouts focus on how elements resize on change in font sizes. *Hybrid* layouts which combine fluid, elastic and fixed layout styles are also common.

Computing a fixed layout from a mockup is straightforward because the positions and sizes of all elements are fixed and hence they can be styled independently. In contrast, to generate a relative layout, we need to consider the relative vertical/horizontal alignment of elements in the page, which may also change locally across the page. Therefore, a more sophisticated approach is needed.

Layout Representation. A horizontal box *HBox* contains child boxes aligned along the x-axis (left-right). The children of a vertical box *VBox* are aligned along y-axis (top-down). We use *HVBox* to denote a box which is either an HBox or a VBox. A cut is called a *guillotine cut* if it breaks a connected area into at least two parts. A vertical (horizontal) guillotine cut is a cut parallel to the y-(x-) axis and creates an HBox (VBox), respectively. An *HVBox partition* P of B is obtained by performing a sequence of vertical or horizontal cuts; P partitions B into a hierarchy of disjoint HVBoxes. The box type of an element n is denoted by n.box.

Most web pages are laid out in a structured manner and their layout can be captured using HVBox partitions. Given a mockup $\mathcal{M}$ drawn on a canvas box C, a *HVBox layout* for $\mathcal{M}$ is defined to be a HVBox partition P of C such that each minimal partition in P contains exactly one UI element from $\mathcal{M}$. Inferring an HVBox layout allows us to model the structure of the page in a holistic fashion; the model captures relative sizes and positions of the UI elements, as well as the vertical/horizontal *flow* of the elements at different locations on the page. This, in turn, enables encoding a relative layout for $\mathcal{M}$ in HTML efficiently.

Grid vs HVBox layouts. Layouts are often modeled using grids.

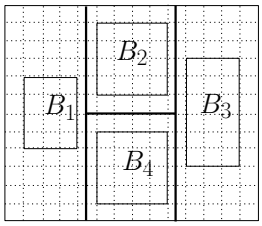

Figure 5: HVBox (solid) vs Grid (dotted) partitioning.

However, grids result in fine-grained layouts which have additional overhead. Consider Fig. 5 showing layout with four boxes B_1-B_4. To model the layout using grids, we need to specify the fine-grained grid shown in dotted lines. In contrast, we only need 3 guillotine cuts with an HVBox partitioning (solid lines). Encoding grids with HTML tables is even worse, e.g., because of redundant markup for empty table cells and harder re-designs [16]. Further, designer must spend additional effort chopping content and aligning boxes with the grid in a WYSIWYG editor [1, 21], which is laborious.

As mentioned earlier, our encoding framework (Fig. 1) consists of two phases: (a) infer an HVBox layout T, (b) encode the layout T into HTML/CSS. Obtaining faithful encodings is non-trivial: both these phases require sophisticated algorithmic machinery and multiple design choices. We now describe the first phase of our framework, i.e., HVBox layout inference.

4. HVBOX LAYOUT INFERENCE

Because mockups contain implicit layout information, we must extract their layout by analysis. An HVBox layout L for a mockup $\mathcal{M}$ may be viewed as a tree T, where the leaves represent the content boxes, an intermediate node is an HVBox, and the root of T represents the top-level box (*body* in HTML). Computing a valid HVBox layout L may involve searching over a large number of possible HVBox partitions. To minimize the search space, we use a greedy search which performs *iterative bottom-up merging* of boxes by choosing a pair of boxes to merge at each iteration. Some of these choices may not lead to a valid HVBox layout. Therefore, the algorithm is capable of backtracking, i.e., if it is unable to proceed by merging further, it backtracks and tries out other possible merge choices. A valid layout L corresponds to a correct sequence of merges (Theorem 1).

The procedure ComputeLayout (Alg. 1) computes the layout tree T for $\mathcal{M}$, using procedures Preprocess, Merge and MergeImplied. It maintains a *forest* of trees F to compute layout: initially, F contains a separate tree for each box in $\mathcal{M}$. The procedure finishes successfully if F contains a single tree T at some iteration. Procedure Merge merges two boxes $b_1, b_2 \in F$ to obtain an HBox (VBox). It first computes the *minimum bounding box* (MBB) for b_1 and b_2, i.e., the smallest horizontal (vertical) box c which contains b_1 and b_2. A new box (tree) $c = (box, b_1, b_2)$ is added to F, where box is HBox or VBox, and b_1, b_2 removed from F. To improve search, a *potential merge set* $\mathcal{P}(b)$ for each box b is maintained which contains all the boxes in F which may merge with b.

Merge forms the core of the layout procedure: it needs to compute adjacent boxes which can be aligned and merged, up to a degree of tolerance. We use the k-d tree data structure to store box coordinates, compute $\mathcal{P}(b)$ and answer box overlap queries efficiently. The procedure MergeImplied computes the merges which are *implied* at each iteration, i.e., for all boxes $b_i \in F$ whose $\mathcal{P}(b_i)$ contains exactly one element, say b_j, the procedure merges b_i and b_j. The procedure continues until no implied merges are possible.

In Alg. 1, the Preprocess method first strips $\mathcal{M}$ to extract the set of hierarchical boxes drawn by user. Because a recursive partitioning is infeasible with overlapping boxes, it first computes a

maximal subset of non-overlapping boxes F[1]. After preprocessing, Merge combines boxes in F into an HVBox, while updating the forest F until a single tree remains in F. Note that this merge may update $\mathcal{P}$ of other boxes adjacent to b_i and b_j and hence trigger other implied merges. After computing implied merges, the algorithm checks if a *conflict* occurred, defined as follows.

A *conflicting* configuration during search is where the set $\mathcal{P}(b)$ for each block $b \in F$ is empty and F contains more than one element. Clearly, we cannot continue merging and obtain a valid layout with the current sequence of merges. Therefore, we *undo* one or more previous choices and continue the search by trying other merge choices. Moreover, we *learn* facts from the conflict into a conflict set $\mathcal{R}$. This set stores all the incompatible merge choices encountered during search and prevents the algorithm from making similar bad choices in the future.

The procedure HandleConflict implements conflict analysis and backtracking. On success, HandleConflict returns $true$ and the algorithm proceeds to find other merge choices. Otherwise, no merge choices will lead to a valid HVBox layout, and the algorithm terminates with a NO answer. Otherwise, the algorithm returns the computed layout tree $T = F[0]$. The procedure can be extended to the case where the mockup contains user-defined containers, e.g., an HTML form, which contain children UI elements. In this case, we check if there exists a parent box p in $\mathcal{M}$ which corresponds to the bounding box b obtained on merging the children UI elements. Then Merge labels the merged node in T with p instead of b.

Readers may observe that our layout algorithm, in spirit, follows the *explore-fail-learn* paradigm of the general DPLL algorithm (cf. [46]) for constraint solving, which is known to solve hard combinatorial problems efficiently in practice. We choose to formulate the approach over boxes directly instead of using a SAT solver so as to exploit the problem structure as well as avoid expensive propositional or quantifier-based encodings.

Procedure ComputeLayout (Mockup $\mathcal{M}$)
  F := Preprocess($\mathcal{M}$)
  MergeImplied() //Update F, $\mathcal{P}$
  while *size of* $F > 1$ **do**
    Choose box $b_i \in F$ and $b_j \in \mathcal{P}(b_i)$
    Merge(b_i, b_j) //Update F, $\mathcal{P}$
    MergeImplied() //Update F, $\mathcal{P}$
    if Conflict() **then**
      r := HandleConflict() //Update $\mathcal{R}$, $\mathcal{P}$
      if $\neg r$ **then return** *NO*
  return $F[0]$

Algorithm 1: Layout Computation.

THEOREM 1. *If a mockup has a valid HVBox partition, then the* ComputeLayout *procedure will compute a valid tree for it.*

Proof Sketch. We prove by induction over the number of horizontal/vertical cuts in the HVBox partition. For a single cut, the mockup is an HBox or VBox with two elements. Merge will obtain the HBox or VBox via a single merging step. Suppose the iterative merging obtains a valid layout for all HVBox partitions with $n-1$ cuts. Consider a partition with n cuts: remove any one cut from it to obtain an $n-1$ cut partition. By hypothesis, iterative merging will compute a correct layout tree T for the latter. Now, the n^{th} cut must separate two boxes b_i, b_j, both of which lie in some leaf l of T, and $b_j \in \mathcal{P}(b_i)$. This is sufficient for Merge to combine b_i, b_j into tree t and replace l in T by t to compute T' for the n-cut partition.

Example. Consider the mockup with boxes 1a-1d, 2, 3, 4 and 5 shown in Fig. 6(a). Starting with a forest consisting of these singleton box-trees, ComputeLayout tries to merge the trees into a single tree iteratively. Fig. 6(b) shows the initial merge set $\mathcal{P}(b)$ for each box b as a graph: each edge denotes that the corresponding pair of nodes may be merged without intersecting with other boxes. The graph shows that, in Fig. 6(a), 1a-1d only merge among themselves, box 3 has two merge choices, 2 and 4, and box 5 has no choices (merging 5 with 4 intersects 3, with 1d intersects 1c). Implied merges are computed first: there are two of them ($\mathcal{P}(2) = \mathcal{P}(4) = 1$) and the algorithm must make a choice. Suppose it chooses to merge boxes 2-3 first. Now, box 4 cannot merge with 3 and there are no more implied merges. Further, suppose boxes 1a-1d are merged among themselves, to form a box labeled 1. Fig. 6(c) shows the resultant configuration. Now, all the boxes 1, 2-3, 4 and 5 have their merge set $\mathcal{P}$ empty. Hence we have a *conflict*. To resolve the conflict, we undo the earlier merge choice 2-3 and record (2-3, 1a-1d) as an incompatible set of merge choices (if we merge 2-3, then 1a-1d cannot be merged). Now, the other merge choice 3-4 from Fig. 6(b) is tried. After merging 3-4, $\mathcal{P}(3-4) = \{5\}$ and $\mathcal{P}(1) = \{2\}$. The procedure computes implied merges: boxes 1-2 are merged and boxes 3-4 are merged with box 5. Finally, 1-2 and 3-4-5 are merged as shown in Fig. 6(d) and we obtain a valid HVBox partition. The final layout tree is shown in Fig. 6(d) with the intermediate HBox (VBox) nodes labeled by H(V).

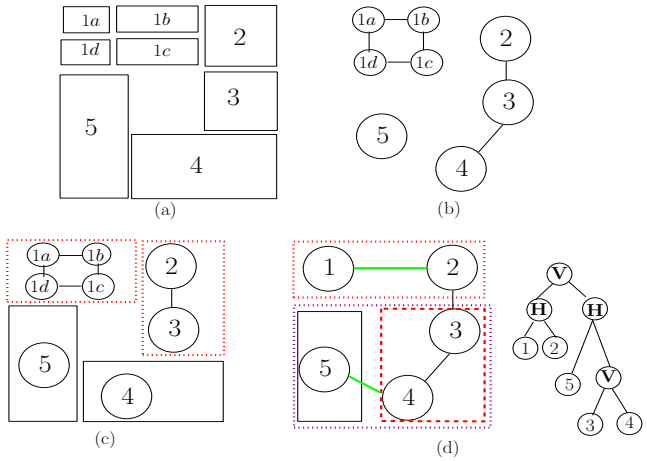

Figure 6: Example: Layout Inference.

Procedure Compact (Tree T)
  repeat
    foreach *node* n *in* T **do**
      if c *is a child of* $n \wedge c.box = n.box \wedge$
      isRemovable(c) **then**
        For each child g of c, make g a child of n
        Delete c from T
  until T *does not change*;

Algorithm 2: Compact the partition tree. For element x, $x.box$ has values HBox or VBox or is undefined.

The tree T computed by ComputeLayout may have redundant HVBoxes, e.g., if a tree node n is an HBox and all its children are also HBoxes, we can *flatten* T by removing each child c of n and attaching the children of c to n. In general, note that every child HBox c of an HBox n is redundant, i.e., all children of c can be made direct children of n without affecting the HVBox layout. Using this observation, Compact (Alg. 2) computes a minimized partition with fewer HVBoxes. We cannot remove an intermediate node n from tree T if it corresponds to an element in the mockup, which is deemed essential by the designer. The isRemovable procedure checks if an element is essential before removing it.

[1]Most designs avoid such boxes but complex designs may contain a few. We handle overlapping boxes at a later stage.

5. HTML/CSS ENCODING

Encoding the inferred layout T in HTML/CSS would be straightforward if CSS supported HBoxes and VBoxes natively. Unfortunately, no such constructs exist in CSS2.1 [3] and as we discussed earlier, using HTML tables makes layout redundant and mockup drawing labor-intensive. Thus, we need to exploit the CSS box semantics, as professional designers do, in order to "hack" CSS to create the desired layout. Two main CSS notions are used for layout in practice: *block formatting contexts*(BFCs) and *floated* elements. The precise definitions of these notions are quite involved [3]: we present a simplified description here to provide intuition.

Encoding HVBoxes. A block formatting context (BFC) is the key layout primitive in CSS: it defines an independent scope for formatting elements contained inside it, oblivious to other page components. A block box b, e.g., *div*, may define a new BFC by using special CSS keywords, e.g., *overflow* or *float*, in the CSS rules for b. By default, blocks in a BFC are laid out from top-to-down, one after another (*flow layout*). So, we can simply encode a VBox in CSS by defining a new BFC (HTML *body* element is a BFC already). The main issue is how to create horizontal flow for HBoxes. CSS *float* properties come to our rescue here. Given a list of child boxes $[b_1, b_2, \ldots, b_n]$ inside an HBox p, we can encode them by *floating* each b_i to the left (or right) inside p. This is specified by adding (*float: left*) for each b_i. Intuitively, this may be understood as an extension of text wrapping, e.g., in LaTeX, to boxes: floating boxes *wrap* next to their previous box sibling. Additional care is needed when we would like to recover the vertical flow: we must *clear* the floats after the final child b_n. This intricate combination of BFC and floats enables nested horizontal and vertical layouts.

CSS Quirks. The description above glosses over many CSS quirks, which our encoding also needs to handle. Consider, e.g., the issue of *collapsing parents*. A floated child b_i does not contribute to parent p's container size in CSS, or in other words, b_i is ignored when sizing p. This behavior, known as the *great collapse* popularly, leads to a bad layout in many cases. Fig. 7(a) shows the same example from Fig. 4(b) except that the *content* box p is highlighted by adding the rule *background: #cbc* to *#content* in CSS. Now, we float children of p using *.mybox* class, leading to the collapse, shown in Fig. 7(b). Interestingly, this was not apparent in Fig. 4(c), but is still undesired. To fix this, we establish a new BFC with p so that p's layout now *owns* its children's layout: add a rule *overflow: hidden* to p's CSS *#content*. Other tricks, e.g., by clearing floats, can also solve this problem.

As mentioned earlier, our encoding should be both fluid and elastic (Sec. 3) and not fixed. Fluidity is essentially obtained by specifying size/margins of boxes in percentages relative to their parent, but involves additional challenges: we must consider how the complex multi-box structure (Fig. 3) interacts with the size/margin properties. Handling padding and borders (in pixels) together with fluidity introduces rounding errors and requires computing tolerant relative values. Many more challenges exist, e.g., collapsing margins, aligning labels with corresponding input elements in forms, etc., discussed later. Overall, automatic encoding into CSS turns out to be quite an *ordeal*. Note that at present, these encoding hardships need to be borne manually by the designer.

5.1 Rule-based Modular Encoding

We wish to hide the aforementioned CSS hardships under-the-hood to reduce designer's burden. To this goal, we propose a modular rule-based architecture for encoding layouts systematically. Our approach automates and unifies multiple adhoc design principles, has a clear separation of sub-tasks and avoids re-computation of intermediate results. This, in turn, allows it to be reusable even as CSS evolves or encoding goals change.

Fig. 8 shows the architecture: boxes represent the key individual modules and the arrows show their mutual dependency. The top modules, MAKE-BLOCK, COMPUTE-RELATIVE and MARK-FLEX compute intermediate results which are required by multiple lower modules, i.e., MARK-FLOAT, MARGIN-FLUID, SIZE-FLUID and SIZE-FLUID-ELASTIC, which encode the alignment, relative margins and the height/width of UI elements. Each module is defined by a set of rules, given in Fig. 9, which recurse over the structure of layout tree T. Specifying modules as rules helps highlight the key logical challenges during encoding and enables an abstract description, decoupled from their application order in a specific implementation.

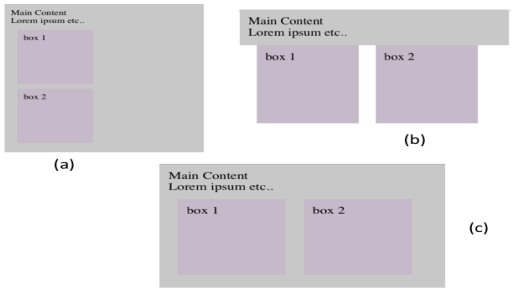

Figure 7: Collapsing parents with floating children.

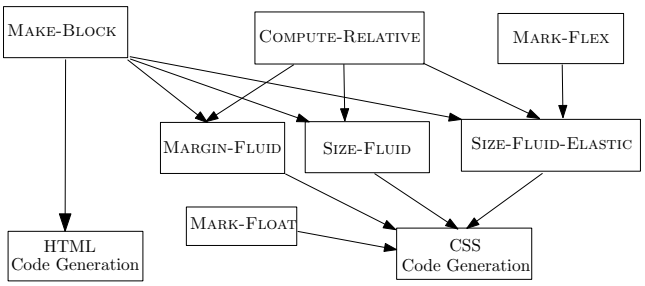

Figure 8: HTML/CSS encoding architecture overview.

Encode structure (HTML). Encoding the structure of the tree T in HTML is relatively straightforward. Each intermediate node in T maps to a *div* element (<*div*></*div*>) which encloses the HTML code for its subtree. The root node maps to the *body* HTML element. The leaf UI elements in the layout tree T are encoded by corresponding HTML tags or widgets [17]. Care must be taken to encode the content of the UI element properties, e.g., all the options of a *select* element must be encoded into HTML.

5.2 Encode layout and style (CSS).

The modules in Fig. 9 encode tree T into CSS in a manner faithful to the original mockup, i.e., preserve relative positions, sizes, alignment and visual properties. Here, variables c, cp, n range over nodes of the layout tree T. chn(n) = number of children of n, $child(c, n)$ ($childK(c, n, i)$) holds if c is some child (i^{th} child) of n in T. HBox(n) or VBox(n) holds if n.box = HBox(VBox).

[MAKE-BLOCK] Recall that only block box elements (cf. Sec. 2), e.g., *div*, *ul*, may be assigned height/width values and establish a BFC in CSS [3]. Therefore, we must convert non-block leaf elements n in tree T to block elements to control their size and ensure proper alignment. However, children of a few block elements, e.g., p cannot be made block [8]. Hence, this module (Fig. 9) recurses down T using the rules canMakeBlock and mayHaveBlockChildren. When it finds an non-block element n ($\neg$ isHTMLBLockElement(n)) which can be converted, it sets property n.toBlk to $true$. This module additionally takes care of elements, e.g., *tr*, which do not allow immediate block children but allow other descendants to be block. The toBlk value is looked up during HTML code generation to generate a *wrapper div* block containing n: this block essentially serves as an alias for n in the other rules which enforce the size and position of n.

For CSS translation, we associate each node n in T with a CSS

object, denoted n.css, which contains properties of n computed by the modules. As discussed above, non-block elements cannot accept height/width/margin properties and hence are wrapped by alias *div* elements. So, to assign CSS properties correctly, we use the $cssObj$ function: if n has a wrapper *div* (n.toBlk is set), then $cssObj(c)$ returns the CSS object for wrapper of c, else it returns c.css. The rules refer to n.css or $cssObj(n)$ appropriately.

[MARK-FLEX] To handle text-containing *flexible* elements, this module marks all such elements and their ancestors with property isFlex = $true$. Note that marking all ancestors is necessary to allow flexible children to re-size freely without overflowing their parents.

[MARK-FLOAT] This module defines how the layout HVBoxes are encoded in CSS. VBoxes are encoded directly; the key question is how to encode an HBox. This module traverses T recursively and assigns (overflow = 'hidden') value to each HBox element and (float = 'left') values to all its children. This floats children to the *left* edge of their parent. Margin computation (see below) takes care of the spacing in-between even if the children are closer to the right edge of the parent.

[COMPUTE-RELATIVE] This module pre-computes the offset and height/width of an element relative to its parent's *content box*. The offsets along four directions between child's and parent's *content* boxes are stored as properties topr, leftr, bottomr and rightr with pixel values. The relative height/width relHeight, relWidth are computed in percentages. This module additionally computes the offsets between the child's *border* box and the parent's content box and stores them as properties topr', leftr', and so on (not shown in Fig. 9). These are required by the MARGIN-FLUID module.

[MARGIN-FLUID] Margin values align the set of children properly inside their parent's container (cf. Fig. 3). To obtain a fluid layout (Sec. 3), we compute margins in percentages (of parent's width). All margins for non-block elements are set to 0 (M5). The margins for the first child c of parent n, given by $childK(c, n, 1)$, depends only on the separation between content box of n and the border box of c (rule M1). For other children of n (rules M2-M4), the left/top margins depend on the right/bottom border of the previous sibling (fields immT, immL) depending on whether n is a HBox or a VBox, respectively. Computing only the left and top margins is sufficient for fluid designs if the height/width of children relative to their parent is fixed, i.e., we can set the right margin of c to 0 and take into account the width of c when computing the left margin of the next sibling of c. This also helps avoid the collapsing margin problem because, e.g., only the top margins are specified for each element in VBox. This assumption, however, breaks down for elastic designs (see below).

[SIZE-FLUID] This module computes the sizes of each box relative to its parents in percentages. The height and width of non-root nodes simply assume the relHeight and relWidth values computed previously by module COMPUTE-RELATIVE. The root and non-block nodes assume 100% of their parent's size. This allows the root node to resize with the browser and all other nodes resize automatically due to their relative sizes. However, fluid sizes are not sufficient for handling flexible elements with text, which may overflow its container on resizing.

[SIZE-FLUID-ELASTIC] Encoding flexible elements which respond to both font and viewport changes gracefully is non-trivial. Note that in fluid layouts, child size depends only on the parent size (*upward* dependency). However, in an elastic layout (Sec. 3), the size of c also depends on the text font size for c, i.e., c must grow as the font size increases. Consequently, the size of all the ancestors of c now also depends on the size of c (*downward* dependency). If we enforce only a fluid layout, i.e., the contained text may overflow the boundaries of the child on font size increase. On the other hand, if c's and its ancestors' size dependent solely on the font size, e.g., by setting the size to *auto*, then c and its ancestors may expand or shrink to an undesired size depending on font size changes. Having both fluidity and *auto* constraints is not possible because it introduces a *circular* dependency between c and its

[MAKE-BLOCK] **Convert *non-block* leaves to blocks**

$$\text{(B1)}\ \frac{child(c,n)\quad b = (canMakeBlock(n) \wedge mayHaveBlockChildren(n))}{canMakeBlock(c) := \ b}$$

$$\text{(B2)}\ \frac{leaf(n,T)\ \neg isHTMLBlockElement(n)\ canMakeBlock(n)}{n.\text{toBlk} := true}$$

$$\text{(B3)}\ \frac{}{canMakeBlock(root(T)) := true}$$

[MARK-FLEX] **Mark elements flexible**

$$\text{(FE1)}\ \frac{hasFlexSize(n)}{n.\text{isFlex} := true} \qquad \text{(FE2)}\ \frac{child(n,p) \qquad n.\text{isFlex}}{p.\text{isFlex} := true}$$

[MARK-FLOAT] **Encode HVBoxes using float**

$$\text{(FO1)}\ \frac{\text{HBox}(n)}{n.\text{overflow} := hidden}\ \frac{child(c,n)\ f = (\text{HBox}(n)\ ?\ \text{left} : \text{none})}{c.\text{float} := f}\ \text{(FO2)}$$

[COMPUTE-RELATIVE] **Offset and Size relative to parent**

$$\text{(CR1)}\ \frac{\neg\ root(n,T)}{n.\text{topr} := n.\text{y} - n.\text{parent.y} \qquad n.\text{leftr} := n.\text{x} - n.\text{parent.x}}$$

$$\text{(CR2)}\ \frac{}{\begin{array}{c} n.\text{rightr} := n.\text{leftr} + n.\text{width} \qquad n.\text{bottomr} := n.\text{topr} + n.\text{height} \\ n.\text{relHeight} := n.\text{height}/n.\text{parent.height} * 100\% \\ n.\text{relWidth} := n.\text{width}/n.\text{parent.width} * 100\% \end{array}}$$

[MARGIN-FLUID] **Encode Margins in CSS**

$$\text{(M1)}\ \frac{childK(c,n,1) \qquad o = cssObj(c)}{\begin{array}{c} o.\text{margin.top} := n.\text{topr'}/n.\text{width} * 100\% \\ o.\text{margin.left} := n.\text{leftr'}/n.\text{width} * 100\% \end{array}}$$

$$\text{(M2)}\ \frac{childK(c,n,i) \quad 1 < i \leq chn(n) \qquad childK(cp,n,i-1) \qquad \text{HBox}(n)}{c.\text{immT} := c.\text{topr'} \qquad c.\text{immL} := c.\text{leftr'} - cp.\text{rightr'}}$$

$$\text{(M3)}\ \frac{childK(c,n,i) \quad 1 < i \leq chn(n) \qquad childK(cp,n,i-1) \qquad \text{VBox}(n)}{c.\text{immT} := c.\text{topr'} - cp.\text{bottomr'} \qquad c.\text{immL} := c.\text{leftr'}}$$

$$\text{(M4)}\ \frac{childK(c,n,i) \qquad 1 < i \leq chn(n) \qquad o = cssObj(c)}{\begin{array}{c} o.\text{margin.top} := c.\text{immT}/n.\text{width} * 100\% \\ o.\text{margin.left} := c.\text{immL}/n.\text{width} * 100\% \\ o.\text{margin.f} := 0, \text{f} \in \{\text{bottom,right}\} \end{array}}$$

$$\text{(M5)}\ \frac{n.\text{toBlk} \qquad o = n.\text{css}}{o.\text{margin.s} := 0 \qquad \text{s} \in \{\text{top,left,bottom,right}\}}$$

[SIZE-FLUID] **Encode Fluid Height/Width in CSS**

$$\text{(SF1)}\ \frac{o = cssObj(n) \qquad n \neq root(T)}{o.\text{height} := n.\text{relHeight} \qquad o.\text{width} := n.\text{relWidth}}$$

$$\text{(SF2)}\ \frac{n.\text{toBlk} \vee n = root(T) \qquad o = n.\text{css}}{o.\text{f} := 100\%, \text{f} \in \{\text{height,width}\}}$$

[SIZE-FLUID-ELASTIC] **Encode Elastic-Fluid Height/Width**

$$\text{(SFE1)}\ \frac{child(c,n) \qquad o = cssObj(c) \qquad \neg n.\text{isFlex} \qquad \neg c.\text{isFlex}}{\begin{array}{c} o.\text{height} := c.\text{relHeight} \\ o.\text{width} := c.\text{relWidth} \end{array}}$$

$$\text{(SFE2)}\ \frac{child(c,n) \qquad \text{HBox}(n) \qquad n.\text{isFlex}\ c.\text{isFlex} \qquad o = cssObj(c)}{o.\text{width} := c.\text{relWidth}}$$

$$\text{(SFE3)}\ \frac{child(c,n) \qquad \text{HBox}(n)\ n.\text{isFlex}\ \neg c.\text{isFlex} \qquad o = cssObj(c)}{\begin{array}{c} o.\text{min-height} := c.\text{height} + \text{px} \\ o.\text{width} := c.\text{relWidth} \end{array}}$$

$$\text{(SFE4)}\ \frac{child(c.n)\ \text{VBox}(n) \qquad n.\text{isFlex}\ \neg c.\text{isFlex} \qquad o = cssObj(c)}{\begin{array}{c} o.\text{min-height} := c.\text{height} + \text{px} \\ o.\text{min-width} := c.\text{relWidth} \end{array}}$$

Figure 9: Rules for CSS Encoding.

ancestors. We therefore have to carefully balance the upward and downward dependencies to obtain a desirable layout.

Our key insight here is that by constraining *only the width* of the elements relative to their parent, we can keep the flexible elements (and their ancestors) unconstrained, while ensuring fluidity of the overall design. The height of flexible elements is left unconstrained (or minimally constrained, if required) so that they can adapt to viewport and font changes. To encode this (cf. Fig. 9, we remove the earlier rule (SE1) which assigns fluid constraints naively and add rules (SFE1-4). For a flexible child c of an HBox n (SFE2), we only set its relative width, leaving height unconstrained. For a non-flexible sibling c' of c (SFE3), we constrain the *min-height* of c' to its mockup-given height in pixels. This prevents c' from shrinking to an undesired size. Ideally, *max-height* for c' should also be set but it wasn't required for our benchmarks.

If n is a VBox n, we leave the flexible element size fully unconstrained, but we constrain the min-height and min-width of its non-flexible siblings (rule SFE4). Recall that MARGIN-FLUID module assumes that fixed relative height/width of children with respect to its parent, which does not hold now. So, we also modify rule (M4) to compute bottom (right) margin values for all children of a flexible HBox (VBox) and right (bottom) margins for the last HBox (VBox) child (rules not shown to save space). Note that our rules avoid circular dependencies as well as achieve elastic-fluid layout.

Font properties. For the UI elements with specific font-size and line-width properties, a separate module encodes these font properties in relative notation by using percentages. This allows changing the font properties throughout by making a single change at the top-level *body* element, without losing the correlation between the font-sizes of different elements.

Code Generation. The HTML code is generated by recursively traversing the layout tree, generating wrapper *div*s, and each element is assigned a unique class name. CSS code is generated for each element using its class name as the selector followed by its CSS properties, e.g., height, width, float, margin, computed using the rules above. A normalizing CSS style file [14] is added in the beginning for ensuring cross-browser compatibility and setting properties like padding, margin and border values to 0 wherever necessary. Identical CSS annotations for different HTML elements are also grouped together in a single CSS class.

Extensibility. Our encoding framework is modular and each module can be refined independently without affecting other modules as long as the interfaces remain the same. For example, we could easily refine rules from module SIZE-FLUID to handle elasticity without modifying the overall flow. Multiple implementations of the same module can co-exist and selected by the designer on-demand, e.g., MARK-FLOAT can be implemented differently using dummy *div*s and clear. Modularity also enables fine-tuning our encoding for issues specific to UI elements or browser behaviors.

5.3 Additional Implementation Challenges

Rounding. Rounding errors during multiple margin and size calculations can cause the child content to overflow its parent. To avoid this, we truncate all values to one decimal position. For images with elastic layout, we set both their *width* and *max-height* to 100% of the wrapper box and do not set the *height* property. More optimizations are possible, e.g., if a node has a single child, *margin: auto; text-align: center;* properties may be used to center the child. Moreover, we can float HBox children to right if necessary during margin computation, and then set right instead of left margins.

User Guidance. Our automated layout inference approach may not be always successful in computing a correct HTML/CSS encoding due to a number of reasons. For example, the mockup may be ambiguous (cf. 1a-1d in Fig. 6) and not capture the designer-intent fully, such that there may be multiple valid merge choice sequences and therefore multiple feasible layouts. Consequently, our HVBox inference algorithm may not infer the layout desired by the user. Different or non-standard implementations of CSS standards may also cause problems. In such cases, user guidance is required. Our tool enables the user, via configuration files, to guide the layout inference and encoding for a particular design, e.g., specify groups of elements which must be merged or must not be merged. The tool provides multiple other options to enable user guidance: float clearing mechanism, fix *body* height in pixels or percentage, specify elements with fixed size in pixels, use a fixed max-width for text-containing boxes, and whether to automatically center a box if it is the only child. Moreover, we can allow the user to fix other style properties, e.g., *position*, *z-index*, in the WYSIWYG editor: these values will be retained in the final design and override the tool-computed values in case of conflict.

Browser Incompatibilities. Our encoding relies on a small set of CSS idioms to maximize cross-browser compatibility. Further, to get rid of the well-known browser inconsistencies due to CSS, we prepend a *normalizing* stylesheet [14] to each generated mockup. Still, the pages may not display correctly in all browsers, e.g., because the browser does not implement the CSS2.1 standards correctly or completely. In such cases, we can use conditional layouts, e.g., we can use CSS conditional comments to detect older versions of the Internet Explorer (IE) browser and apply different CSS rules to correctly display the page. We believe that this is a reasonable compromise to support legacy browsers in the face of rapid adoption of newer browsers (supporting CSS2.1 standards to a large extent and even CSS3 substantially) and modern markup technologies. Finally, we note that browser compatibility cannot be guaranteed statically and needs testing specific to targeted browsers. Nevertheless, our tool saves considerable manual effort in page design by automating layout inference and HTML encoding systematically. Our framework, being modular and systematic, can also be re-targeted to other existing or new technologies, as they arrive. We have also provided the designer several knobs to tweak the designs to match the user-intent as well as deal with limitations of CSS. Also, we observe that many hurdles of cross-browser compatibility and limited CSS2.1 expressivity will be overcome once CSS3 is standardized and all browsers support it fully.

Overlapping Boxes and Dynamic Layouts. Recall that our framework discards the overlapping boxes before inferring layout. In most cases, we observed that the designers intend these boxes to be located at a fixed position relative to the viewport. Therefore, we add these boxes back to the design at the original fixed positions with original sizes using *position* property. We do not support dynamic rearrangement of layout as of now. However, our float-based encoding of HBoxes together with min-width values partially overcomes this drawback: if the viewport size is too small, children of HBoxes will rearrange automatically to the next row.

CSS3 [5] is an in-progress standard for styling web documents and contains much better support for specifying HVBoxes and relative layout than CSS2.1. However, the standard is still in flux and different web browsers support it to different degrees. Our modular encoding framework (Fig. 8) can be reused for CSS3 also by instantiating the modules with CSS3-specific rules, many of which do not change. Further, our mockup-based compilation approach allows transparent extension to generate CSS3 code behind-the-scenes, without burdening the designer.

6. TOOL DESIGN AND EVALUATION

We implemented and evaluated our approach using an open-source mockup builder *Maqetta* [11] being developed at IBM. Maqetta supports design of mockups of both desktop and mobile applications by dragging-and-dropping UI elements on the editor canvas, specifying CSS properties for user elements. Maqetta generates HTML on-the-fly when mockup is drawn. It provides two kinds of default layout modes, *flow* and *absolute*, for designing applications. In flow layout, the elements are placed vertically one-after another. To align elements horizontally, the designer must specify it using CSS floats manually. In the absolute layout, the elements are fixed to specific coordinates, thus allowing arbitrary placement

Benchmark	Description
master-detail	Navigate the items of the master menu and view the details on the same screen
column	Hierarchical tabular data navigation
search	Display search results for a particular query
filter	Refine data search results further
form	Standard form UI
canvas	Canvas object for drawing graphics
dashboard	Show aggregated information at a glance
spreadsheet	View and edit tabular information
wizard	Guides user through a complex work-flow
qa	Select among multiple questions
parallel	View inventory list of similar items
interactive	Combination of forms with result display

Table 1: Benchmarks corresponding to standard UI patterns.

of UI elements on a web page. However, Maqetta generates a *fixed* layout where elements do not respond to change in viewport sizes.

Our tool takes an input a Maqetta mockup specification drawn in absolute layout mode and generates a faithful replica of the mockup in HTML/CSS, which has a relative layout. The designer specifies only the key UI elements constituting a web page, without worrying about the enclosing boxes which determine the page layout. The height, width and position of the elements along with margins and the relative alignment of elements are extracted from the visual design and need not be specified separately. The designer, however, needs to specify properties like padding and border widths for the elements separately, which are hard to specify visually.

Our tool is wholly implemented in JavaScript, both on the client-side (Maqetta) and server-side (node.js application [13]). The mockup in Maqetta is serialized and transmitted to the server, which processes it, infers layout, encodes it to HTML/CSS, and dumps the corresponding HTML/CSS files to the disk. We modified the Maqetta source to add a button [*Create Layout*]: clicking the button serializes the mockup to JSON and sends it to the server using an AJAX call. Note that our back-end can also be moved into the browser for creating layouts if necessary.

6.1 Evaluation

To show that our encoding is effective, we require benchmarks covering the wide array of possible web designs. Unfortunately, no representative benchmark set exists to evaluate a tool like ours. Selecting designs from the top Alexa pages does not make sense because they are in no way representative of the possible web layouts. Therefore, we constructed a benchmark set consisting of web pages which capture various representative design patterns found across web. We used multiple popular design sites, including Yahoo! Design Pattern Library [20] and *ui-patterns.com* to collect these benchmarks and curated them manually to avoid similar or duplicate patterns. Table. 1 shows the representative patterns and their descriptions. We drew mockups corresponding to these patterns in Maqetta and performed layout and code generation for them. Note that these designs closely imitiate real websites and are quite complex and hence may contain up to a few hundred elements. Obtaining correct fluid-elastic layouts needed a couple of iterations: the encoding procedure had to be refined multiple times before we got it correct. We verified the correctness of the generated encodings using Firefox 16 (Mac Snow Leopard 10.6.8, Windows 7), Chrome 23 (Mac Snow Leopard 10.6.8, Win 7) and IE9 (Win 7). We also used browser tools to visualize the output for different form factors.

Our tool takes only a few seconds to encode each benchmark and all the encoded files represent faithful replicas in HTML of the original mockups. As opposed to the fixed layouts generated by Maqetta, the generated designs are fluid, i.e., they resize with the browser (cf. Fig. 2) gracefully. Moreover, they are elastic, i.e., the boxes containing text expand or shrink based on font-size changes, without destroying the overall fluidity of the design.

Analysis. Table 2 shows details of the benchmarks and their encoding in HTML. The first column shows the number of user-drawn UI elements (boxes), which vary between 36 to 225 for our benchmarks. Although the number of neighbors (nearby elements) for boxes may go up to 18, the set of mergeable neighbors (merge set, cf. Alg. 1) remains small (2 or 3) because the merged elements quickly begin to overlap. This reduces the merge choices and enables the algorithm to converge fast. The third column shows the total number of boxes after layout inference and the layout tree height (cf. Fig. 6); the fourth column shows the same values after compaction (cf. Alg. 2). Compacting the tree is essential: it removes a large number of redundant boxes introduced by the layout inference and reduces the tree height significantly, thus reducing the size of the final HTML/CSS code. The next five columns show the number of attributes computed for layout and size/spacing. The most benefits in terms of designer effort reduction is due to these attributes, in particular height, width and margins. Note that in absence of our tool, the designer will have to make careful calculations of relative size and margin values to obtain fluid layouts. Besides, she will need a deep understanding of BFCs and floats to encode the HVBox layout correctly. Also, note the difference between number of height and width attributes computed. In order to obtain elastic layouts (Sec. 5.1), our tool omits constraining the height wherever possible; however, widths are always constrained to ensure fluidity. With our tool, the designer need not put in additional effort to ensure the balance between fluidity and elasticity.

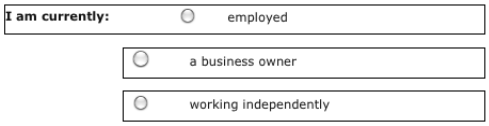

Figure 10: Incorrect box merge during layout inference.

Layout Inference. We did not encounter any conflicts in these benchmarks during layout inference because implied merging (cf. Alg. 1) often avoids them successfully. However, an incorrect set of elements were merged together sometimes, e.g., in Fig. 10, instead of merging all the checkbox-label pairs in a single VBox, the algorithm merges the label "I am currently" with the top checkbox-label pair. To solve this, we provide manual guidance (Sec. 5.3), by marking the checkbox-label pairs to be merged. While such manual effort is unavoidable in general (hard to obtain unambiguous mockups), better symmetry detection heuristics can detect such element groups automatically.

Fine-grained Tweaks. Most of the encodings generated by our tool are high-quality replicas of the original mockup. Also, they adapt gracefully on reducing the viewport size or increasing font size: columns shrink in width as viewport size reduces while the text splits into multiple lines to accommodate the smaller width. Owing to complexity of CSS and incompatible implementations, a few fine-grained issues remain. In some cases, e.g., *filter*, *master-detail*, the *button* does not fill its wrapper *div* element (see MAKEBLOCK): this is fixed by setting the min-height of the button in pixels and removing the min-height constraints from its parent wrapper. Same fix is applied for the HTML *select* elements also. These fixes make the designs less fluid, but they are unavoidable with the current less-expressive CSS standards. To obtain correct image sizes, setting the max-height of the image to 100% is not sufficient in some cases. Then, we also fix the image wrapper's height in pixels. Input elements, e.g., checkbox or radio buttons, do not align properly in some cases. This is solved by removing the constraints *height: 100%; width: 100%* relative to their wrapper *div*s. In some cases, we want the child element to fill the parent *div* even if the latter's size is not specified. This is fixed by adding padding to expand the child. Note that these fine-grained refinements can be also included as special rules, invoked under user-guidance, in our extensible framework. Single word labels begin to overlap with other elements on resizing to small widths. Finally, making the viewport too small or the font too large may distort the design, requiring

Benchmark	#Drawn Boxes	#Neighbors /#Merge Set	Inf. #Boxes /Tree Ht.	Min. #Boxes /Tree Ht.	#Inferred Attributes					File Size(KB)	
					float	overflow	height	width	margin	HTML	CSS
master-detail	83	13/3	125/11	91/6	64	20	60	125	80	11.5	13.8
column	225	18/2	318/14	232/7	129	33	81	212	111	18.5	40.1
search	62	17/2	104/10	65/4	50	10	46	96	56	7.4	10
filter	87	10/3	129/12	96/7	56	18	59	117	82	12	12.3
form	36	13/3	58/9	45/6	20	9	27	47	38	8.6	4.7
canvas	72	12/3	110/11	86/6	46	16	63	112	77	9.9	13
dashboard	55	7/3	77/10	67/8	28	14	34	66	53	6.9	7
spreadsheet	158	7/2	219/15	159/5	102	25	82	186	93	30.1	29.3
wizard	62	7/2	89/8	66/4	40	17	42	75	61	6.6	9
qa	55	14/3	97/8	81/6	42	21	37	80	60	9.8	11.2
parallel	65	8/3	105/9	88/7	24	12	49	75	63	10.7	12.6
interactive	70	18/3	108/11	80/5	42	15	56	83	63	6.6	8.8

Table 2: Tool statistics for the benchmarks in Table 1. See Analysis in Sec. 6.1.

dynamic layouts. In spite of these issues, our experiments demonstrate that high-quality flexible layouts can be indeed obtained automatically from mockups, with minimal manual effort.

Experience with drawing mockups. Drawing each mockup took the authors about an hour on average. Note that the authors are not experts in UI design or Maqetta; we believe experts may be able to draw mockups faster. A few time-consuming factors were setting color, border and font properties, obtaining images, aligning elements inside a parent box or keeping them equi-distant. Some of these issues can be addressed by allowing a tolerance during size calculations and better symmetry detection, both of which are part of our planned tool improvements. We believe that in absence of our tool, a few hours of additional manual effort would be required per mockup to align, size and layout elements properly.

7. RELATED WORK

Automated Document Formatting has been studied extensively (see [29, 38] for surveys) and most approaches model the problem as a constraint optimization problem. Much work has focused on *micro*-typography specific to text-dominated documents (cf., e.g., [34]). For *macro*-typography, i.e., layout of objects, constraint-solving based methods [22, 29, 25] are used: they take input alignment constraints between UI elements, their sizes, etc. as input, and solve these constraints to output a layout, i.e., coordinate values of each UI element on the canvas. The work in [35] uses constraint solving to generate multiple mockups similar to the original one. All these methods infer pixel locations of UI elements as opposed to inferring a regular box/grid structure in which the elements can be embedded. The latter problem, which is our focus, is completely different from the earlier one - techniques for solving the earlier problem do not apply here. Moreover, constraint-solving based methods assume a dedicated layout engine and cannot use an off-the-shelf browser for layout directly. Instead, our goal is to leverage the browser's layout engine fully with no augmentations. Several JavaScript-based engines perform dynamic layout inside the browser; however, using JS for layout unnecessarily burdens the application and fails to exploit the native layout engine.

The limited expressivity and complexity of CSS for web design is a known problem [37, 32, 27, 45, 30]. The work in [44] asks for debugging tools for CSS and proposes solutions to assist CSS development in authoring tools while [32, 31, 47] propose compilation from more expressive style languages to CSS to improve productivity and avoid bugs. None of these approaches, however, infer CSS layout rules from visual descriptions. Constraint cascading style sheets [23] augment CSS with expressive arithmetic constraints relating sizes of elements and font sizes, but require a modified browser layout engine for supporting it. Techniques to assist CSS-based development by eliminating redundant CSS rules [39] or by performing sophisticated property checking and coverage analysis [27] for CSS rules have been proposed.

Modern WYSIWYG editors [1, 12, 11, 2] allow convenient CSS authoring with drag-and-drop features and employ their own layout engines to enable page visualization. However, they generate only fixed or non-hierarchical layouts automatically. Complex layouts must be specified manually by the designer. Some tools [1] now enable grid-based design also; however, additional manual effort is required to make these designs elastic and fluid simultaneously.

Rectangular partitioning (see [41] for a survey) has been studied extensively. Many of these problems reduce to *covering* problems where boxes may be reordered to minimize cost. In contrast, boxes may not be reordered in our setting. Computing an optimal HVBox layout is an instance of the problem of *hierarchical binary tiling* [33], whose complexity is polynomial in the grid size, but with a high exponent value. Therefore, instead of adopting the usual dynamic programming solution, we propose an efficient backtracking based procedure, which consumes less memory and tries to find one solution instead of the optimal one. Guillotine partition for table layout has been investigated in [26] using dynamic programming. Our work obtains a partition for arbitrary set of boxes on a canvas (not just tables). The work in [36] uses machine learning to transfer page design from one page to another.

Popular *Content Management Systems* (CMSs), e.g., Drupal [6] or Wordpress [19], natively provide the designer with a set of fixed layout templates to create pages. Many systems, e.g., zengrids [21], allow *grid-based* fluid layout specification using modern CSS extensions [15, 10]. A few CMS plugins, e.g., Sasson for Drupal, enable flexible layouts in the CMS, again based on re-sizing grids. Such frameworks, however, require user to explicitly fit the contents into the grid by specifying the location and the number of grids taken by a box. Further, they require learning new styling primitives of the framework. In contrast, with our tool, the user can freely draw UI elements in a WYSIWYG editor while our tool takes care of the relative layout completely. Further, to our knowledge, no tool infers a hierarchical layout from the mockup, which captures local flow of elements besides their relative size/spacing.

8. CONCLUSIONS

We presented a method to systematically encode mockups drawn in a WYSIWYG editor into flexible (fluid and elastic) layouts. Our method first infers a hierarchical HVBox layout for the mockup and then encodes the layout with modular rules into HTML/CSS. We believe that ours is the first disciplined effort to solve the challenging problem of automatically encoding mockups in HTML/CSS. The ability to generate flexible HTML encodings from mockups automatically will allow quicker development of the application for multiple devices at once, besides reducing the learning curves among novices. We do not generate fully dynamic layouts currently; future work will build upon the current framework to obtain such layouts, possibly with more involved constraint solving. Our modular framework can also be extended to generate optimized, cleaner CSS [27] using modern CSS extensions [15, 10, 47]. We also plan to augment Maqetta to enable user feedback directly in Maqetta, as opposed to using a textual configuration file, and carry out usability surveys. We believe that automated compilation will enable the tool to exploit evolving technologies, including improved layout techniques e.g., CSS3 [5], SASS [15], and re-enable adoption of mockup-based UI development.

Acknowledgements. We would like to thank Satish Chandra, Vyas Sekar, Partha Datta and the anonymous reviewers for helping us improve the presentation of this paper.

9. REFERENCES

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

[2] Balsamiq. http://www.balsamiq.com/.

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

[4] Cisco visual networking index: Forecast and methodology, 2011-2016. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html.

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

[6] Drupal - open source cms. http://drupal.org/.

[7] Gecko - mozilla | mdn. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko.

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

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

[10] Less: The dynamic stylesheet language. http://lesscss.org/.

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

[12] Microsoft Expression Web 4. http://www.microsoft.com/expression/products/Web_Overview.aspx.

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

[14] Normalize.css: Make browsers render all elements more consistently. http://necolas.github.com/normalize.css/.

[15] Sass - syntactically awesome stylesheets. http://sass-lang.com/.

[16] Techniques for web content accessibility guidelines 1.0. http://www.w3.org/TR/WCAG10-TECHS/#tech-style-sheets.

[17] Unbeatable javascript tools - the dojo toolkit. http://dojotoolkit.org/.

[18] The webkit open source project. http://www.webkit.org.

[19] Wordpress: Blog tool, publishing platform, and cms. http://wordpress.org/.

[20] Yahoo! design pattern library. http://developer.yahoo.com/ypatterns/.

[21] Zen grids: a responsive grid system built with compass and sass. http://zengrids.com/.

[22] 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.

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

[24] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock. Decision-theoretic user interface generation. In *AAAI*, pages 1532–1536, 2008.

[25] 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.

[26] G. Gange, K. Marriott, and P. Stuckey. Optimal guillotine layout. In *Proceedings of the 2012 ACM symposium on Document engineering*, DocEng '12, pages 13–22, 2012.

[27] P. Geneves, N. Layaida, and V. Quint. On the analysis of cascading style sheets. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 809–818, 2012.

[28] V. Grigoreanu, R. Fernandez, K. Inkpen, and G. Robertson. What designers want: Needs of interactive application designers. VLHCC '09, 2009.

[29] 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.

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

[31] M. Keller and M. Nussbaumer. Cascading style sheets: a novel approach towards productive styling with today's standards. In *WWW*, pages 1161–1162, 2009.

[32] M. Keller and M. Nussbaumer. Css code quality: A metric for abstractness; or why humans beat machines in css coding. In *QUATIC*, pages 116–121, 2010.

[33] S. Khanna, S. Muthukrishnan, and M. Paterson. On approximating rectangle tiling and packing. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, SODA '98, pages 384–393, 1998.

[34] D. E. Knuth. *Digital Typography*. Cambridge University Press, New York, NY, USA, 1997.

[35] Ali Sinan Köksal. Live tiles from end-user mockups. CHI 2012.

[36] 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.

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

[38] S. Lok and S. Feiner. A survey of automated layout techniques for information presentations, 2001.

[39] A. Mesbah and S. Mirshokraie. Automated analysis of css rules to support style maintenance. In *ICSE*, pages 408–418, 2012.

[40] Leo A. Meyerovich and Rastislav Bodík. Fast and parallel webpage layout. In *WWW*, pages 711–720, 2010.

[41] S. Muthukrishnan, Viswanath Poosala, and Torsten Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 236–256, 1999.

[42] M. W. Newman and J. A. Landay. Sitemaps, storyboards, and specifications: a sketch of web site design practice. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, DIS '00, pages 263–274, 2000.

[43] F. K. Ozenc, M. Kim, J. Zimmerman, S. Oney, and B. Myers. How to support designers in getting hold of the immaterial material of software. CHI '10, pages 2513–2522, 2010.

[44] Vincent Quint and Irène Vatton. Editing with style. In *ACM Symposium on Document Engineering*, pages 151–160, 2007.

[45] D. Reed and J. Davies. The convergence of computer programming and graphic design. *J. Comput. Sci. Coll.*, 21(3):179–187, February 2006.

[46] K. A. Sakallah and J. Marques-Silva. Anatomy and empirical evaluation of modern sat solvers. *Bulletin of the EATCS*, 103:96–121, 2011.

[47] Manuel Serrano. Hss: a compiler for cascading style sheets. In *PPDP*, pages 109–118, 2010.

[48] B. V. Zanden and B. A. Myers. Automatic, look-and-feel independent dialog creation for graphical user interfaces. CHI '90, pages 27–34, 1990.