Programming by Manipulation for Layout

Thibaud Hottelier
UC Berkeley
tbh@cs.berkeley.edu

Ras Bodik
UC Berkeley
bodik@cs.berkeley.edu

Kimiko Ryokai
UC Berkeley
kimiko@ischool.berkeley.edu

ABSTRACT

We present Programming by Manipulation, a new programming methodology for specifying the layout of data visualizations, targeted at non-programmers. We address the two central sources of bugs that arise when programming with constraints: ambiguities and conflicts (inconsistencies). We rule out conflicts by design and exploit ambiguity to explore possible layout designs. Our users design layouts by highlighting undesirable aspects of a current design, effectively breaking spurious constraints and introducing ambiguity by giving some elements freedom to move or resize. Subsequently, the tool indicates how the ambiguity can be removed, by computing how the free elements can be fixed with available constraints. To support this workflow, our tool computes the ambiguity and summarizes it visually. We evaluate our work with two user-studies demonstrating that both non-programmers and programmers can effectively use our prototype. Our results suggest that our tool is 5-times more productive than direct programming with constraints.

Author Keywords

Programming by demonstration; layout editing; constraint-based layout

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous; D.2.1. Requirements/Specifications: Tools

INTRODUCTION

Visual layout is the art of arranging visual elements in a pleasing manner. Layout spans multiple application domains, including the layout of documents; the layout of GUIs; the layout of websites, which is a hybrid of document and GUI layouts; and the layout of data visualizations. Both non-technical users and expert programmers design layouts, by using WYSIWYG editors or by writing code that customizes an existing layout library. Either way, creating a layout entails fixing the sizes and positions of some visual elements and specifying, in some manner, the rules for computing sizes and positions of the remaining elements.

Over the years, many alternative ways of specifying layout have been proposed. (See Hurst *et al.* [9] for a recent overview of the field.) Constraints are arguably the most widespread and successful programming technique. For example, the foundations of TEX are laid upon constraints. CSS, the ubiquitous web template language, also relies on constraints, although in a more restricted and indirect manner [9].

Data visualizations are among the hardest layout problems, in part because the data may have a recursive (tree) structure and/or the visual layout is not "boxy." Today, the task of building data visualizations consists of crafting a tailored layout engine for a dataset. The current state-of-the-art, general purpose frameworks [6, 4, 7]—*e.g.*, Protovis and its successor D3—require the advanced technical expertise of seasoned programmers. As such, they are inaccessible to many potential users, such as non-programmer scientists.

Our solution, *Programming by Manipulation (PBM)*, streamlines layout specification by allying user demonstrations with guided exploration of the layout design space. Targeted toward non-programmers, PBM is designed to prevent users from getting stuck in either ambiguities or conflicts (contradictions), the two symptoms of bugs when programming layouts with constraints. Before we detail our solution, we summarize the programmability challenges posed by constraints.

Programming with Constraints

Constraint-based layouts are powerful and versatile (see Related Work). By stating properties of the layout directly, constraints promise to yield a precise and predictable layout specification. However, manipulating constraints directly can be tedious and error-prone. Specifically, programmers must carefully navigate between two hazards: ambiguities and conflicts. Ambiguities arise when we do not state enough constraints of the goal layout (under-specification) allowing multiple distinct layouts to satisfy the constraint system. The first solution found by the solver is unlikely to be the intended one. Worse, the selected solution might be different each run, causing non-determinism. As such, ambiguities make the resulting layout unpredictable for users. However, by stating too much (over-specification) we risk introducing conflicts, *i.e.*, inconsistencies among constraints. When a conflict occurs, there exists no layout satisfying all constraints. Our user-study shows that resolving such conflicts can be challenging, even for experienced programmers. In practice, the solver is often allowed to drop some constraints, sometimes based on a priority hierarchy, until the system admits one or more solutions [2, 26].

Ambiguities are commonly alleviated by casting layout as an optimization problem. If at most one layout maximizes the utility metric, the ambiguity is removed. Leaving aside the difficulty of capturing layout esthetics with a mathematical metric, optimization does not fully address the problem. For

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UIST 2014, October 5–8, 2014, Honolulu, HI, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3069-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2642918.2647378

instance, it is well known that optimization-induced "spring-effects" result in unexpected layouts [26], forcing designers to twiddle with constant parameters by trial and error. Furthermore, ambiguities are reintroduced when conflicts are handled by dropping constraints, because there may be alternative ways to drop constraints, each leading to a distinct layout. This choice falls back upon the solver, which does not have adequate information to make an educated guess, even with priorities attached to constraints[1].

Ultimately, ambiguities and conflicts have the same consequences for users: the resulting layout may be unpredictable and may appear to be chosen arbitrarily. The only certain method for determining the effects of constraints is to run the solver and examine its output. This limitation motivated the programming of constraints by demonstration.

Programming by Demonstration

The advent of Programming by Demonstration (PBD) gave rise to GUI builders. They enable users to express layout by example, from which the necessary layout constraints are inferred automatically. By lowering the level of discourse to concrete visual entities (widgets), away from abstract positioning rules, demonstrations make layout programming accessible to a wider audience. For visual domains such as layout, a natural form of demonstration is a paper and pencil sketch. However, users' drawings contain small errors and imprecisions: they cannot be interpreted literally by PBD systems. For this reason, GUI builders adopted a constructive approach: instead of drawing the entire layout at once, users demonstrate step by step, by progressively adding widgets onto a canvas. However, even with demonstrations, the central issue remains: ambiguities and conflicts creep in during demonstrations, for example when a new widget cannot be inserted without breaking a constraint on existing widgets[2]. When a conflict or ambiguity occurs, users have no other recourse than diving into the constraints to resolve them manually, it may be a challenging task for someone who does not program.

We conjecture that the root causes behind the difficulty of programming with constraints—ambiguities and conflicts—have not been fully addressed. There has been exciting recent work in this area [25]. Most notably, ALE has introduced a language fragment (ALE excluding manual constraints) that is free of both ambiguities and conflicts. In terms of programmability, this is an ideal language. However, some layouts can be difficult to express. For instance, to center a widget globally, users need to manually add constraints from outside this fragment, which may reintroduce conflicts.

Layout for Visualizations

With the understanding that ambiguities and conflicts are the central programmability issues, let us now look at the specific needs of data visualization layout. Consider a biologist (a non-programmer) trying to define a phylogenetic tree. Precise control over positions and alignment is crucial, because the layer in which each node is drawn has biological meaning: it determines when two species branch off. The number of possible tree layouts is very large: by considering only positional aspects of tree layouts such as the overall architecture (flat, radial, flower-like); the layering strategies; the space allocation strategies between both parent/children and siblings, we count over a hundred possible designs. It is unlikely that our biologist will find the required layout in a library of prepackaged visualizations, such as ManyEyes [22]. Therefore, we propose to steer the exploration of designs by manipulating the layout of the sample document.

Finally, data visualization has requirements distinct from GUI: layouts are non-boxy and recursive, and datasets inevitably change and grow. Therefore, layout engines must be generic enough to be reusable for new, updated data. Moreover, scientific datasets can be massive; users must be able to demonstrate the layout semantics on a small subset of the data and then run the resulting layout engine on the full dataset.

We summarize our design principles in the following four points:

1. The language of constraints must be rich enough to capture a wide class of data visualizations, including recursive and non-boxy ones.
2. The system must be resistant to the small imprecisions present in drawing-based demonstrations.
3. Ambiguities and conflicts must either be ruled out or be explained at a level of discourse understandable by non-programmers.
4. Users must be able to demonstrate the desired layout on a small subset of their data. The demonstration must generalize to other datasets.

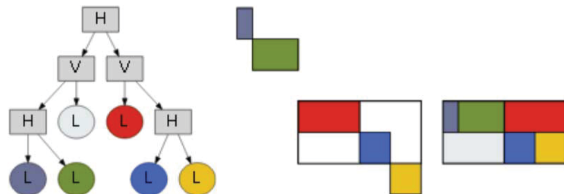
Programming by Manipulation (PBM)

We propose Programming by Manipulation (PBM), a new example-driven programming paradigm, based on guided exploration of the space of layout configurations. We cast layout as a *satisfaction* problem, avoiding the reliance on an optimization utility function. To help our designer select constraints just sufficient to yield a single solution, we develop a manipulation methodology that guarantees the absence of conflicts and actively steers the user away from ambiguities by explaining them visually and proposing potential resolutions. Our manipulation explores a design point opposite to ALE, which rules out ambiguities and explains conflicts.

[1] With CSS, text overflowing the borders of a container is a classic illustration of conflict resolution not matching the designer's intent [15]. This phenomenon occurs with only two boxes: Box A containing the text with a preferred width of 300px, and its decoration, box B, set to half as wide as the window. The designer would like A to be contained inside B. In CSS, this is expressed indirectly by making A a child of B. When the user resizes the window to 500px, CSS will overflow the text of A out of B. If B has a visible border, the resulting layout is unlikely to please.

[2] We illustrate this problem with an example inspired by ALE [12, 25]. Using a GUI builder, we add two text boxes next to each other and horizontally justified on a window 240px wide. We set the width of each text-box to 100px. By adding a third widget to the same row whose width must be at least 50px to be displayed properly, a combo-box for instance, we create a conflict. The sum of width of our three widgets is over 240px. When faced with this situation, XCode silently drops the width constraint of the text-boxes. ALE extracts the relevant conflicting constraints to help the designer understand and eventually repair the constraints manually.

Creating a data-visualization with PBM proceeds in two main stages (Figure 1a). First, the user selects the broad class of the layout, *e.g.*, a tree, by building a sample document. This document is constructed by instantiating building blocks from a library. Each block is flexible, in that it encapsulates a large set of constraints that are individually activated based on configuration switches. For instance, most blocks provide one constraint per alignment strategy, and a switch controls which strategy the block should use. By choosing which blocks to use, the user already defines the principal characteristics of the layout: is the layout flow-based (*e.g.*, a tree) or guillotine-based (*e.g.*, a treemap); is it radial or Cartesian? The sample document is a partial specification of the layout; it is partial because the configuration switches have not yet been set. This configuration process happens in the second step, where the user *browses* through possible configurations, each yielding a different layout. The user does not toggle switches directly. Instead, she steers the exploration by manipulating the layout of the sample document by dragging blocks.

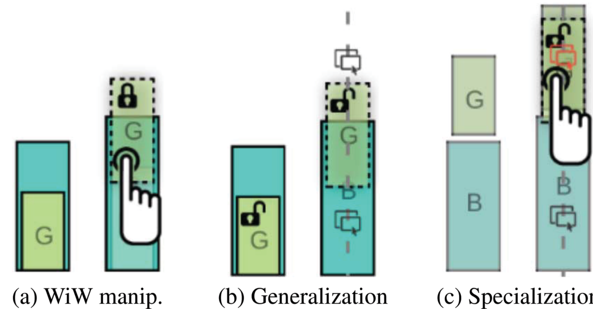


(a) First the user create a sample document, a tree of blocks (left). Then, starting from an arbitrary initial configuration (middle), she manipulates the layout (of the sample document) until obtaining the final (desired) configuration (right).

(b) The final configuration can now be applied to larger datasets (documents).

Figure 1. Creating a treemap with PBM. The 3 stages (a) to establish a layout specification, which is reusable (b).

Our tool—the PBM manipulator (PBMM)—allows the user to perform two moves: (i) *generalizing* the layout by disabling some constraints, thereby introducing ambiguities; or (ii) *specializing* the layout by enabling new constraints, effectively resolving ambiguities. Generalizations are expressed with a new type of demonstration which tolerates imprecisions in manual demonstrations, so-called *what is wrong (WiW)* manipulations (Figure 2a). Like StopThat interactions [14], WiW manipulations are negative demonstrations. Instead of indicating what the desired layout should be, the user points out one incorrect aspect of the current layout by dragging a block away from its constrained position. To interpret such manipulations, PBMM only considers the "direction" of the WiW manipulation as opposed to the final locations of the displaced blocks. Behind the scenes, PBMM determines which constraints to toggle off to allow the block to move in the demonstrated direction (Figure 2b). It does so by choosing the ones for which the newly introduced ambiguities best align with the direction of the WiW manipulation. We have essentially sidestepped the interferences originating from the inherent imprecisions of user drawings. Specializations are expressed by letting the user choose one layout feature among a range of options (Figure 2c). Only safe specializations (conflict-free) are proposed.



(a) WiW manip. (b) Generalization (c) Specialization

Figure 2. Three steps to modify a barchart from overlapping bars to stacked bars: By dragging the green box upward, past its snapping radius (a), we break its vertical positioning constraint. To explore alternative layouts, freedom is introduced in response to our WiW manipulation by the PBM system (b). Both green boxes, two instances of the same block, are now vertically free. Finally, by dropping one of the green boxes onto the topmost specialization site (red icon), we fix the vertical position of both green bars at once by adding a new constraint to the configuration (c).

Unfortunately, we cannot shield users from both ambiguities and conflicts: starting from a deterministic layout of the sample document, to hop to the next deterministic layout, we must toggle switches to both disable (generalize) and enable (specialize) constraints. Depending on which operation is performed first, the user will be confronted at the intermediate point with either an ambiguous or conflicting layout. We argue that conflicts are more difficult to explain than ambiguities. Ambiguities can be conveyed by examples, by showing a range of possible layouts. Whereas, to explain conflicts, one must spell out why something is impossible, a task intrinsically more difficult to visualize. For this reason, we chose to prevent users from ever creating conflicts and to have them tolerate ambiguities by explaining them with visual cues. We condense all ambiguities into a few "axes of freedom" which convey not only which blocks are unconstrained, but also which layout aspects of those blocks remain to be specified. For instance, a block might have a fixed horizontal position while being unconstrained vertically.

Once the layout has been configured, we know which constraints to use: we have established the complete specification of the layout. We can finally turn the configuration into a layout engine (*i.e.*, an executable layout template) which takes the document as a runtime input and lays it out (Figure 1b).

Contributions

Programming by Manipulation brings the following benefits: First, conflicts can no longer arise; specialization only lets users enable constraints which resolve ambiguities previously introduced via generalizations. At worst, users browse back to a previously seen layout. Second, we explain ambiguities with a visual summary understandable at a glance. Finally, our exploration-centric approach enables users to discover for themselves which layouts are feasible given the language

of constraints and settle for a close enough approximation if necessary [11].

This paper makes the following contributions:

1. The Programming by Manipulation[3] paradigm targeted toward non-programmers for visual domains such as data-visualization.

2. A new type of demonstration—What is wrong (WiW) manipulations—which is resistant to users' imprecisions, inherent in drawing. Instead of sketching the desired layout, users steer the exploration by pointing out what they would like to change on a given layout. Only the direction of the manipulation is interpreted by PBMM.

3. Two user-studies, the first one showing that non-programmers can design interesting visualizations using our PBM tool. 10 out of 11 participants completed all five visualization tasks. The second study demonstrates that proficient programmers are more productive with PBM than with conventional constraint programming. With PBM, programmers needed on average one fifth of the time and three times fewer attempts to complete the same tasks.

RELATED WORK

Programming by Manipulation builds on the foundations laid by constraint-based layout systems, GUI builders, and the recent work on fully automatic layout inference.

Constraint-based Layout

Constraints have been used in many languages to specify layout [5, 19, 13, 18, 1, 12, 10, 9]. Much work has focused on expressibility and solving efficiency. (In contrast, we are concerned primarily with programmability, by preventing conflicts and explaining ambiguities.) Typically, layout has been phrased as an optimization problem by either maximizing a utility metric or satisfying as many constraints as possible. We chose to cast layout as a satisfaction problem, with the following trade-offs: satisfiability is a simpler problem in term of computability; rich constraints such as polynomials, which are common in visualizations, become tractable. Satisfaction also enables a deeper level of analysis: we leverage the power of SMT-solvers to prevent conflicts, summarize ambiguities, and efficiently compute both generalizations and specializations.

GUI Builders

With GUI builders, users can construct user interfaces graphically by progressively adding widgets to a canvas [17, 18, 20, 23]. Each time a widget is added, new layout constraints fixing its position are inferred, sometimes with the help of semantic snapping [8]. More advanced systems produce flexible GUIs which adaptively resize to occupy the space available [25]. Naturally, GUI builders are tailored toward UI boxy or tab-stop layout [12]; it is unclear whether these techniques can be adapted to recursive layouts common in data visualizations, such as a radial tree.

Most GUI builders delegate the resolution of conflicts and ambiguities to users. Our user-study suggests that this is a challenging task. Recent work has focused on this programmability challenge: ALE [25] is a layout editor which guarantees that the layout is well-defined (non-ambiguous) and explains conflicts by computing the maximum satisfiable set of constraints. ALE also defines a safe, conflict-free fragment of the layout language (one without manual constraints). This comes at the cost of some expressiveness; for instance, centering globally is not possible. We took the opposite approach and chose to rule out conflicts but tolerate ambiguities. We believe that ambiguities (and their resolution) are easier to convey to users than conflicts. We condense all ambiguities into a summary: a set of "axes of freedom" understandable at a glance by non-programmers. ALE and PBM have orthogonal approaches to how a layout is constructed. We start from a full, complete but incorrect specification, and progressively adjust it by enabling and disabling constraints. In contrast, ALE starts with an empty specification and progressively fleshes it out by inferring more constraints as widgets are added to the layout.

Automatic Layout Inference

Fully automatic methods for layout generation have been studied as well. Layout can be inferred from topological descriptions [24], or directly from user-drawn mock-ups [21]. In the latter work, a subdivision of the space expressed as a tree of vertical and horizontal dividers is extracted from a single demonstration, a mock-up. This hierarchy is then encoded with CSS rules which can be laid out by a web browser. Since a single mock-up may not be a sufficient specification of the layout, user guidance is invoked to deal with the ambiguity. This user guidance takes the form of configuration options which include manually fixing some of the subdivision steps.

PROGRAMMING BY MANIPULATION

This section provides a detailed overview of layout by manipulation, using a phylogenetic tree as a running example (Figure 3). The goal for our user is to establish the core aspects of layout, such as position, size, alignment, and margins. In the next paragraph, we take a step back to explain why we think our exploration-centric workflow is a crucial feature for PBD systems such as this one.

Our early prototypes performed rather poorly. With neither exploration nor manipulation, users were asked to sketch the desired layout by repositioning all layout elements in one comprehensive demonstration. The combination of constraints inferred rarely produced the desired layout. In our informal observations, we saw that users were left perplexed, knowing neither what they did wrong nor how to improve their demonstration. This direct approach failed because users are unaware of which layouts are expressible with the constraints embedded in the sample document. This is a common flaw of PBD systems [11]. The target program, as represented in a user's mind, is often not expressible. Interestingly, there often exists an equivalent program or a close approximation which is expressible and for which users would settle if they could discover it [11]. This observation led us to our exploration-centric approach.

To create any visualization, we first need to construct a sample document. In a second step, we will configure this document by directly manipulating its layout. Concretely, since we cast

[3]Try our online demo at **http://pbm.cs.berkeley.edu**.

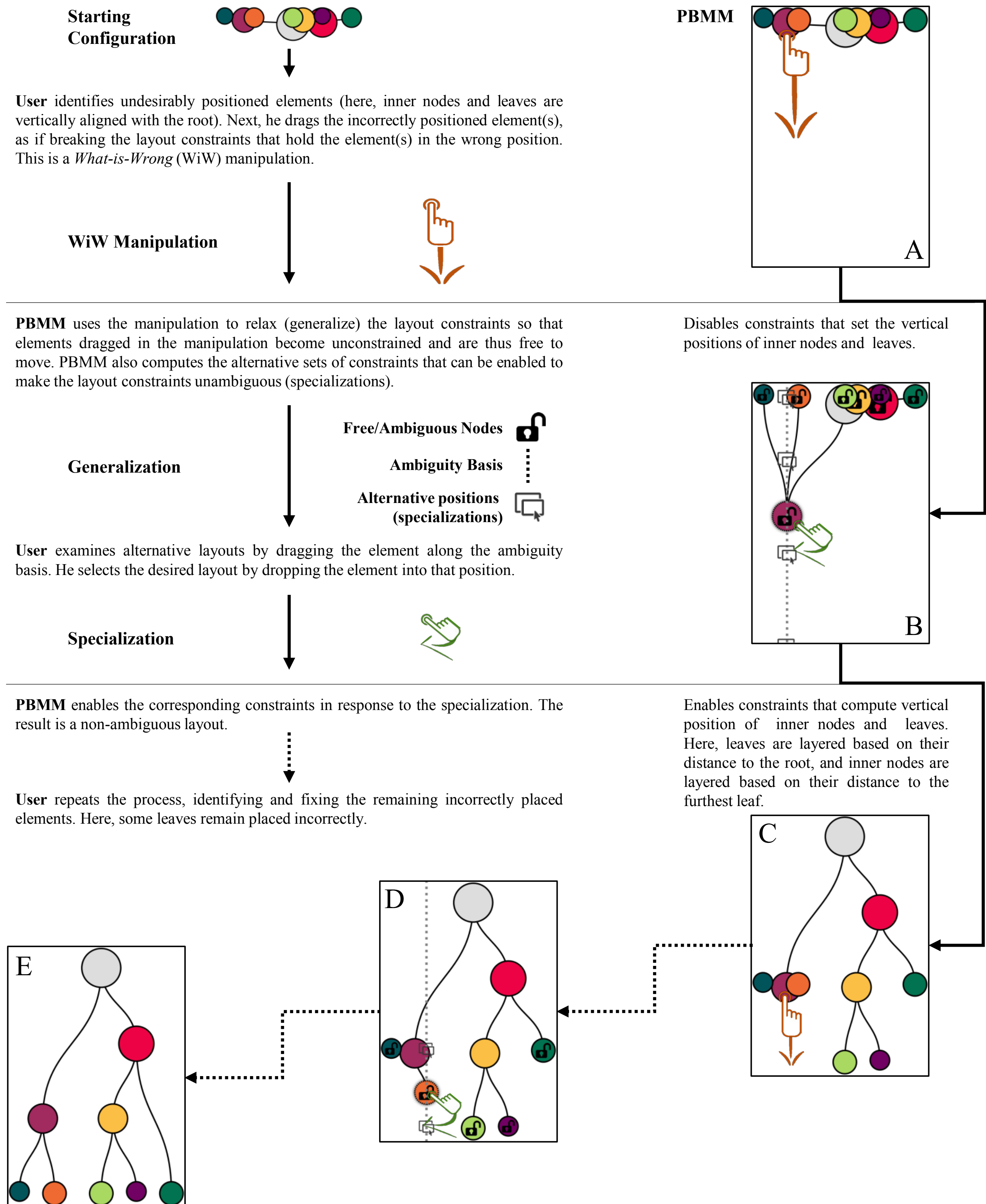


Figure 3. The five configurations explored to establish the node layering of a tree. The hand icon illustrates the manipulation performed by the user. Configurations B&D are ambiguous; their axes of freedom are shown only for the selected node.

layout specification as a satisfaction problem, we must find a combination of constraints leading to a single, unique layout. This combination of constraints constitutes our "layout configuration". Once established, we can reuse the same configuration to create other documents which will share the same layout properties. In CSS terminology, a layout configuration would be called a template.

Creating a sample (unconfigured) document

To create a sample document, the (layout) designer selects blocks from a library and nests them: a *document* is a tree of instances of blocks. By choosing which blocks to use, the designer is already painting the broad strokes of the layout: a barchart and a tree are built from radically different blocks. For our phylogenetic tree, we nest instances of three blocks: *TreeRoot* is the root of our sample document, inner nodes are instances of *TreeNode*, and the leaves are *TreeLeaf*. The sample document must be representative of the type of documents to be supported by the layout configuration. In practice, we found that documents with about 10 to 20 nodes are most useful. A tree with a single node does not provide enough information. However, our biologist's full dataset of over one hundred nodes for a phylogenetic tree has too many entities, making manipulation difficult.

Blocks

Blocks are crafted from constraints by an expert programmer. They have flexible layout behavior controlled by configuration switches that enable or disable individual constraints. It is the role of manipulation to configure these switches. Each block contains both attributes (*e.g.*, sizes and positions) and constraints. Some attributes are known constants, for instance the size of an image, while others need to be computed at runtime. The constraints defining a block range not only over its own attributes but also over those of its neighbors in the document hierarchy.

Since blocks are reusable across many visualizations, we collect them in a library. Each block also bundles an English description of its function for designers. Other frequently used blocks include horizontal/vertical dividers for guillotine layouts (*H/VDiv*), grouping boxes (*HBox, VBox, HVBox*) for box-based layouts, floating elements for flow-layouts (*FloatBox*), as well as various containers. We describe the language constraints behind blocks in the next section.

Layout

Under the hood, a document is a constraint system composed as a conjunction of all enabled constraints. As such, each *configuration* (of switches) yields a different constraint system. The set of all possible configurations forms the *configuration space*. Given a configuration, the *layout* of a document is a solution to its constraint system. In other words, a layout is an assignment of values for each document's attributes, such that all enabled constraints are satisfied. Depending on how many layouts exist for a document, we distinguish three kinds of documents: (i) a document is *deterministic* if it admits exactly one layout; (ii) a document for which there exists no layout is inconsistent: some of its constraints are *conflicting*; and (iii) a document which admits more than one layout is *ambiguous*. We compute the layout of a configured document by solving the corresponding constraint system. Modern solvers [16] can handle documents with hundreds of blocks in less than a second. Finally, once the document is laid out, it can be passed to a renderer for display.

Demonstrating the layout configuration

To establish the finer aspects of the visualization, the designer explores the configuration space in search of the configuration which yields the best layout of the sample document. We built a tool supporting this exploration-centric workflow: the PBM manipulator (PBMM), devised to help designers finding an interesting layout quickly, even in huge configuration spaces. PBM turns conventional demonstrations upside down: Instead of directly demonstrating the goal, designers highlight one layout aspect (*e.g.*, horizontal alignment) they would like to change by dragging one block away from its constrained position. We call such manipulations "what is wrong" (WiW) manipulations.

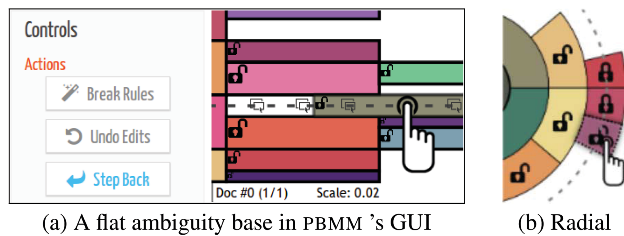
PBMM presents the layout of the sample document according to the currently active configuration. Figure 4a shows the user interface. The exploration always starts from an arbitrary configuration that yields a deterministic document. Then, by manipulating the layout itself, the designer can make a step in the direction of his choosing, which enables him to hop from configuration to configuration. To steer the exploration, the designer either (i) points out an incorrect layout feature; or (ii) chooses an alternative layout from among a range of options. Figure 3 illustrates the exploration process on our phylogenetic tree. In four manipulations, we establish the desired tree layering.

The designer's manipulations are translated into two types of "moves" through the configuration space. One move *introduces* ambiguities and the other one *resolves* them:

- *Generalizations* introduce ambiguities by switching off one[4] constraint currently enabled, effectively weakening the constraint system of the sample document. By toggling off one constraint, we move to a new configuration which admits a superset of the layouts of the current configuration. Generalizations are expressed with WiW manipulations: the designer highlights incorrect aspects of the layout by dragging blocks to displace them from the position constrained by the current configuration. Generalizations are triggered by the "Break Rules" button (Figure 4a).
- *Specializations* resolve ambiguities by strengthening the constraint system of the current configuration. To do so, we switch on one disabled constraint, which brings us to a new configuration admitting a subset of the current configuration layouts. To specialize a layout, designers choose one layout from a list of alternatives.

To browse configurations effectively, designers need to understand the nature of the configuration space: they need to know both "where they are" and "where they can go". While removing constraints (generalization) is always possible, adding constraints (specialization) can create conflicts. As such, PBMM

[4]Constraints are actually switched on or off in groups to handle interdependencies and subsumption. To simplify the presentation, we assume that only one constraint is toggled after each step.



(a) A flat ambiguity base in PBMM 's GUI (b) Radial

Figure 4. Two examples of ambiguity bases. Ambiguous (free) blocks are indicated with unlocked icons. PBMM displays the ambiguity base of the selected block with axes of freedom (dashed-lines). Here, the bases of each selected block are constituted of a single axis. To enable users to "feel" constraints, each block moves freely along its axes of freedom, but resists displacement in other directions by snapping to its axes. We also show PBMM's user interface (a). The "Break Rules" button triggers generalizations.

must also convey which constraints can be added safely to specialize the current configuration. This translates into two responsibilities for PBMM: explaining ambiguities and proposing potential resolutions.

Introducing Ambiguities by Breaking Constraints

To control the introduction and resolution of ambiguities, designers need to know not only which blocks are free (*i.e.*, not fully constrained) but also which aspects (*e.g.*, height or horizontal position) of the blocks are free. A naive solution would be to display each and every possible layout of the sample document. Unfortunately, this is impractical: an ambiguous configuration may admit a very large, sometimes even unbounded, number of layouts. For example, in the barchart illustrated in Figure 2, we remove the vertical stacking constraint of the green bars through a generalization step. As a result, their vertical positions become completely unconstrained, yielding an unbounded number of ways to lay out the green bars. This example illustrates the need to synthesize all ambiguities present in a document into a brief summary, understandable by designers at a glance.

We visualize the ambiguity of a layout with dashed lines, which show which blocks are free and how they are free to move. We call these dashed lines *axes of freedom*. More technically, we summarize the ambiguity with an *ambiguity base*, which is an algebraic base of the space of all admissible layouts of the sample document. The ambiguity base is the same concept as the base of the solution space of a system of equations in linear algebra. In essence, we condense all admissible layouts into the set of independent dimensions. To translate this base into a graphical summary understandable by designers, we display one admissible layout, augmented with axes of freedom, one per dimension of the ambiguity base. PBMM highlights free blocks with unlocked icons. When the designer selects a free block, PBMM displays its axes of freedom. For ambiguity dimensions related to positional attributes, we represent each with a dashed line. Similarly, for attributes related to sizes of elements, we overlay a double arrow mimicking familiar resizing icons. To do so, PBMM understands the visual semantics of common block attributes. Figures 4a and 4b show two ambiguous configurations and their respective axes of freedom, both on Cartesian and polar layouts.

As a bonus, we can use the ambiguity base to make browsing more intuitive through semantic snapping [8]. For instance, blocks which are dragged beyond the layouts admissible by current configuration should resists the designer's action by snapping back to a legal position. As a result, designers can "feel" constraints by manipulating blocks. This also enables us to explain generalizations with the following UI metaphor: by dragging a block past its snapping radius, the designer is "breaking" constraints.

It remains for us to discuss how we compute the ambiguity base. Since our constraints can be both non-linear and non-equational, computing the ambiguity base precisely by cylindrical algebraic decomposition would be prohibitively expensive. Instead, we compute an over-approximation: we determine the subset of the document attributes which can take more than one value under the current configuration. Each such attribute becomes one independent dimension of the ambiguity base. Our over-approximation ignores all relationships between attributes forming the ambiguity base: some of these attributes can be interdependent. For instance, the size of an image might be unconstrained with the exception of its aspect ratio: $4 * \text{height} = 3 * \text{width}$. Such relationships are lost by our approximation: we will consider both height and width to be independent dimensions. In essence, we over-approximate the precise ambiguity base with an enveloping base of higher dimensionality.

Proposing Resolutions (Specializations)

After the designer breaks some constraints, she proceeds to remove the ambiguity by introducing other constraints. Our tool proposes safe resolutions by computing which constraints can be enabled without creating a conflict. Each of these constraint yields a safe specialization of the current configuration

To communicate available specializations to the designer, we represent each of them as a point in the ambiguity space. Given a free block, we mark the points on its axes of freedom where the block would be positioned if that specialization were chosen. Our interface uses semantic snapping to emphasize these specialization points [8]. The designer chooses one of them by drag&dropping a free block onto one of its specialization points. For example, in Figure 2b, the green bars are vertically free. The underlying block behind both green bars embeds multiple vertical alignment constraints. In this simple case, each green bar could be positioned, relative to its respective blue bar, below, above, or vertically aligned along its bottom edge. Each of these positions is marked on the axis of freedom of each green bar, providing a visual enumeration of the potential resolutions. Behind the scenes, each of these positions constitute a specialization of the current configuration. To select one, the designer drags and drops one green bar onto a marked site, as illustrated in Figure 2c.

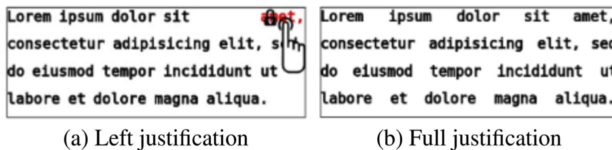
IMPLEMENTATION

First, we present an overview of the language of constraints used by the expert programmer to construct blocks. Then, we explain how we architected PBMM to be independent of the constraints used and to support new blocks without modifications. Finally, we detail our technique to compute generaliza-

tions and explain how we select which ambiguity to introduce based on the WiW manipulation.

Language of Constraints

To capture a wide class of layouts, we support rich constraints including non-linear ones. For instance, polynomials are frequently used to capture ratios of relative sizes. Under the hood, we rely on the SMT bit-vectors theory [3] which provides an expressive set of primitive constructs from which we build our constraints. Figure 5 illustrates the versatility of our blocks with an example of flow-layout with justification.



(a) Left justification (b) Full justification

Figure 5. We illustrate the expressibility of our constraints by capturing text-layout. Both line-break insertions and word-spacing are specified and computed as part of the layout. On the left figure, the user is making a WiW manipulation to go from left justification to full justification.

Our constraints are non-directional (also called bi-directional), they leave the flow of computation unspecified. Non directionality enables greater flexibility and reuse: for example, the expert programmer can fix aspect ratio of an image while leaving open whether the height is computed from the width or vice-versa. The flow of values will be indirectly decided by the designer when she chooses a configuration.

The same constraints are reused across many blocks: for instance, all boxy blocks implement the CSS box model which defines margins, borders and padding. To avoid duplicating such concepts in every block, the library supports composable modules of constraints called *traits*. Beside box models, we have traits for concepts such as horizontal/vertical alignment, grouping, spacing, justification, guillotine dividers, etc. We show below one trait setting up a polar coordinate system. It is used by all our radial layouts (*e.g.*, Figure 4b).

```
1  mandatory
2  trait Polar2Cart(x, y, angle, radius) {
3      x = radius * cos(angle)
4      y = radius * sin(angle)
5  }
```

While most constraints are enabled by configuration switches, there exists a few mandatory ones for essential features such as coordinate systems, as shown above.

To give the flavor of our language, we present some of the constraints available to control the horizontal alignment of the first child of a box.

```
1  optional trait HAlignFirstChild(width) {
2      children[0].left = 0
3      children[0].right = width
4      children[0].right = children[1].left
5      children[0].left = children[1].right
6      children[0].right = 0
7      ...
8  }
```

The configuration switches are implicit. Equal symbols signify equality, not assignment, since our constraints are nondirectional. The first pair of constraints aligns the left/right edge of the (first) child with the corresponding edge of its parent. The next pair places the child to the left/right of its sibling. The last pair of constraints place the child just outside its parent, on the left/right of it.

PBM Manipulator (PBMM)

PBMM has to function out-of-the-box with any blocks written by the expert programmer, and do so without requiring any modification. In essence, PBMM must be constraint-agnostic. The responsibilities of PBMM can be summarized in three computational tasks: Given a sample document together with a configuration, PBMM must (i) compute the layout and the ambiguity base; (ii) compute which specializations are safe (*i.e.*, conflict-free); and (iii) generalize the current configuration given a WiW manipulation.

We accomplish these three tasks by encoding the entire configuration space as defined by the sample document in the SMT bit-vector theory. We used Z3 [16] as our solver. PBMM always starts from a deterministic configuration. We iteratively query the solver for stronger (more switches toggled on) configurations until we find a deterministic one. After that, all computations are performed on-demand, in response to manipulations. Conveniently, our satisfiability approach to layout lets us cast all three PBMM tasks as satisfiability queries. The encoding of the first two tasks are straightforward and have been omitted for space reasons. However, we detail our technique to select which ambiguities to introduce when generalizing configurations.

First, we enumerate all configurations with one less switch toggled on than the current one. This is our set of candidate generalizations, from which we will select the configuration whose ambiguities are most in line with the WiW manipulation. For the sake of explanation, we assume that only one block was displaced by the designer. Consequently, we abstract the WiW manipulation into one vector, capturing the direction of displacement. Then, we rank candidates by counting how many dimensions of their ambiguity base align with the direction vector. Intuitively, we are selecting the principal components of the direction vector expressed in terms of the ambiguity base. More technically, we compute the dot-product between the direction vector and each dimension of the ambiguity base, both normalized to unit length. We consider that the direction vector aligns with an ambiguous dimension if their dot-product is greater than $\frac{1}{\sqrt{n}}$, where n is the dimensionality of the ambiguity base.

EVALUATION

We evaluate our new methodology—PBM—for designing data-visualizations, together with our prototype implementation along the following two axes:

1. Can non-programmers successfully use PBMM to design data visualizations?

2. Can proficient programmers also benefit from PBM by increasing their productivity with the help of PBMM?

Task	Completion		Time [s]	Steps
Barchart A	11	(100%)	17 ± 19	3.6
Barchart B	11	(100%)	64 ± 30	8.4
Icicles	10	(91%)	60 ± 88	8.5
Treemap	11	(100%)	137 ± 55	14.8
Tree	11	(100%)	64 ± 28	5.4

Table 1. Non-programmer results. The columns indicate the number of participants who successfully completed the task; the median time taken in seconds with the standard deviation; and the average number of steps to the goal.

To investigate these two questions, we conducted two user-studies. In the first, we asked non-programmers to configure five data visualizations using PBMM. To answer the second question, we performed a within-subject study on seasoned programmers. We asked them to complete the same five visualization tasks both with PBMM and with an interface mimicking standard constraint programming.

Non-Programmers

We recruited 11 participants (3 males, 8 females, ages 22 to 39) either students or staff from outside the engineering disciplines, largely from the Biology and Linguistics departments. Participants were selected for their lack of formal training in programming. When shown a picture of an icicle graph and asked whether they could program a layout template producing this type of visualization, all participants answered no.

Each session proceeded as follows: Participants were first introduced to PBMM by a 10 minute long, written tutorial, culminating in a simple exercise. Each participant was tasked with creating five visualizations: two barcharts, one icicle layout (Figure 4a), one treemap (Figure 1), and a custom tree layout (Figure 3). These tasks were chosen to showcase the applicability of our method to a variety of layouts, while offering a gradual increase in complexity. Each task consisted of a short introduction motivating the visualization, followed by an illustration of the goal layout. To complete each task, candidates had to produce the goal layout in 10 minutes or less using PBMM.

All participants but one solved each of the five tasks within the time limit. One participant was not able to complete the icicle graph. Results are summarized in Table 1. The treemap is a particularly interesting case: Participants found creative, unexpected ways to complete the task with 8 unique paths through the design space to the goal layout. The shortest path goes through 7 configurations, whereas the longest explores 19, indicating that PBM supports a range of ways to configure a template and accommodates many different thinking processes.

Programmers

To make a fair comparison with manual constraints programming, we focus on the significant aspects of programming, such as resolving ambiguities and conflicts, while abstracting away irrelevant factors like language syntax. To do so, we built a second programming tool which mimics the relevant part of programming with constraints. Instead of typing code, participants toggled GUI switches to enable/disable constraints. In essence, we have reduced the task of constraint programming to finding a set of constraints leading to the desirable layout. We refer to the mock-up tool as the "button" tool.

The interface of the button tool is divided in two: The top half displays the current layout. Users can scroll and zoom in/out, but no other interaction such as dragging an element is possible. The second half is a table of toggle switches controlling constraints. The table has one row per block. Each row contains all the constraints pertaining to one layout element. Columns organize constraints by category, such as "horizontal alignment" or "height computation." Within each cell, each switch is labeled with simplified pseudo-code of the constraint it toggles. If a conflicting set of constraints is enabled, the button tool reports that the selected constraints cannot be satisfied, and no layout is displayed in the top half. The button tool does not provide a debugging aid for identifying conflicting constraints such as the maximum satisfiable subset or the unsat core. However, to explain ambiguous layouts, the button tool does provide the same visual aids as PBMM: the tool shows one possible layout augmented with axes of freedom representing the base of ambiguity for each partially constrained block.

We recruited 16 participants (13 male, 3 female, of ages between 22 and 30), students and staff from engineering departments, mainly Computer Science. All participants had taken at least one CS class and had been programming for at least 3 years. When shown an illustration of an icicle graph, all participants but one claimed they could write a layout template producing this type of visualization.

The programmer study is a within-subject experiment: every participants used both the button tool and PBMM to solve the set of five layout tasks twice. We reused the same five tasks from the non-programmer study. To compensate for learning effects, half of the participants started with the button tool, and half with PBMM. The setup for this study was similar to the non-programmer setup. Participants first read a written, 10 minute long tutorial introducing the first tool, then did a warm-up exercise, and then solved the five layout tasks using the first tool. They then repeated this process (both tutorial and tasks) with the second tool. Finally, we interviewed participants for 10 minutes about which tool they found to be more effective and improvements they would make to either of the tools.

To compare the productivity of participants with each tool, we measured the following indirect indicators: time taken and the length of the path in the design space from the start layout to the goal. Each step in the path corresponds to a configuration which was reached, either by demonstrations or by toggling constraints with switches.

All tasks but two were completed within the 10 minute time limit. One participant could not complete the treemap, and another did not finish the tree, both while using the button tool. We performed an ANOVA of completion times with task and tool as independent factors. The times were log-transformed to make the distribution closer to a Gaussian. We observed a strong main effect of the tool ($F = 345$, $p \ll 0.001$), and significant effect of the task ($F = 72$, $p \ll 0.001$). Since the tasks were specifically chosen to be gradually increasing in

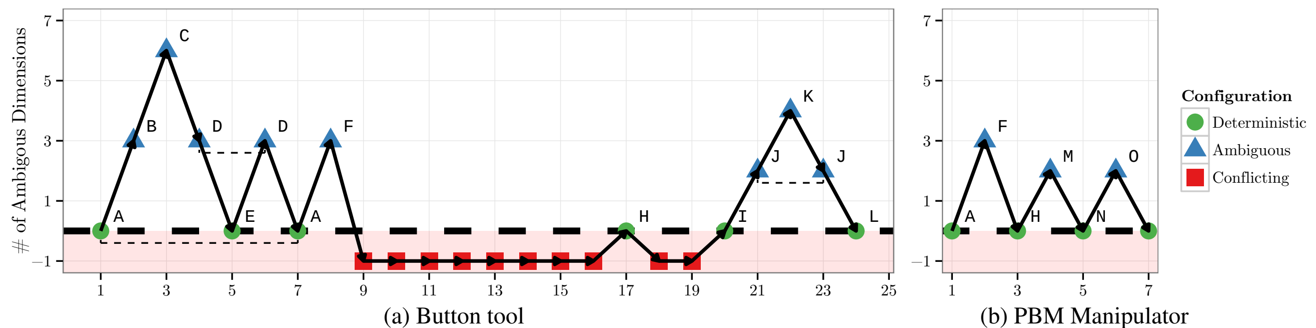


Figure 6. Paths through the configuration space on the icicles task from the same participant using both tools. Each configuration is uniquely identified by a capital letter. A thin dashed-line highlights configurations explored twice due to backtracking. Configurations along the x axis (0 ambiguous dimension) are deterministic. Conflicting configurations are represented with a negative number of ambiguous dimensions.

difficulty, this was expected. PBMM increased the speed of programmers by a factor ranging from 2.5 (Barchart A) to 10.6 (Barchart B). Across all tasks, the median speed-up was 5.3. To analyse the effect of the tool on path lengths, we used a Wilcoxon signed-rank test. We found that the manipulation tool required fewer steps through the design space than the button tool, with strong confidence ($V = 3236$, $p \ll 0.001$). Here again, we observed that paths are approximately 3.6 times shorter on average across all tasks with PBMM.

Discussion

Our first user-study demonstrates that non-programmers can successfully design data visualizations using PBMM, while the second study shows that programmers would also be more productive with PBMM when programming constraints. It is important to note that, in our experiments, the button tool provided instantaneous feedback. The consequences of toggling constraints were immediately visible. In practice, the situation is often worse; programmers must wait for the compilation-execution cycle to finish before seeing the results of their modifications, thereby increasing the time cost of making changes. Consequently, in practice, longer paths to the goal layout are more detrimental to productivity, and the ability of PBM to quickly converge on the goal becomes more relevant.

We have combined the results from both studies to compare the difference in productivity between programmers and non-programmers using PBMM. Non-programmers took on average 53% longer than the subset of programmers who started with PBMM. This is to be expected, since programmers are more familiar with concepts such as constraints: they are able to build a mental model of the inner workings of our tool faster than non-programmers. We argue that a 53% increase in time spent is a small price to pay to enable non-programmers to accomplish tasks which were previously out of reach.

To further understand how participants used each tool; which actions led to dead ends, where users spent time thinking; and where they got stuck; we have examined in detail the traces from programmers with each tool. Figure 6 shows two such traces, one per tool, taken by one participant on the moderately difficult icicle chart. The two traces we have chosen are typical of what we have observed on this task. Note that this particular participant started with PBMM. Let us start with the trace from the button tool. At the beginning, this participant got lost in highly ambiguous layouts and backtracked twice (steps 6 and 7), in effect revisiting the same configurations again. To recover, he eventually backtracked all the way back to the starting point. Then, he started exploring layouts in another direction but got stuck on a conflict (steps 9–16) shortly afterward. It took him eight attempts and a large amount of time—more than two thirds of the total time—to resolve the conflict. Toward the end of the trace (step 23), this participant was deceived one more time by ambiguities, causing him to backtrack again before finally reaching the goal.

Let us now look at the second trace, from PBMM. Interestingly, our participant took a completely distinct path through the design space: Only the start and goal layout engines are common to both traces. Not only did PBMM prevent our participant from creating conflicting configurations, but it also kept our participant in a portion of the configuration space with lower degrees of ambiguity. Recall that the same visual cues (axes of freedom) are used by both tools to explain ambiguities. But even with those aids, understanding what is and is not constrained in layouts with high degrees of ambiguities remains difficult. Highly ambiguous layouts tend to overwhelm users with too much information. Consequently, users are more likely to add an undesirable constraint by mistake in resolving ambiguities. When such mistakes are corrected, the same configuration is explored twice, thereby creating a backtracking step. This "lost in ambiguities" phenomenon highlights the importance of steering users towards layouts with few ambiguities. By proposing possible resolutions for each dimension of ambiguities, PBMM encourages users to settle ambiguities immediately after their introduction. Our participant dealt with at most three degrees of ambiguity, versus six with the button tool. As a result, he never had to backtrack from an erroneous specialization.

In the interviews concluding each session of the programmer study, all but one participant stated they would use PBMM rather than the button tool if given the choice. The one participant who preferred the button tool stated that "the button tool was more challenging thus more fun". Participants expressed frustration with debugging conflicts with the button tool. A common request was to disable (grey out) buttons which would trigger a conflict if toggled. These comments

reinforce our belief that addressing ambiguities and conflicts is essential to making constraint programming more accessible.

On the negative side, participants from the programmers study reported feeling a "lack of control": they would have liked to see how layout engines are modified by their manipulations and which constraints are added or removed. We designed the user interface of PBMM with non-programmers in mind: constraints are completely hidden beneath the UI. For technically-literate audiences, we are considering optionally displaying the layout engine code and using animations to highlight the changes created by each manipulation.

CONCLUSION

We presented Programming by Manipulation, a new methodology for specifying layout with constraints targeted at non-programmers. With PBM, users steer the exploration of layout designs by directly displacing blocks of a sample document. With such manipulations, users can break constraints and subsequently introduce new ones. PBM focuses on programmability and addresses the two principal sources of bugs with constraints: conflicts can no longer arise and ambiguities are explained with a visual summary.

Participants from our user-study seem interested in combining PBM with document authoring so that the sample document can be edited while specifying layout. Our participants were also very enthusiastic about using PBMM to customize CSS templates. This boils down to expressing CSS with constraints, which has been partially done [1]. If one could capture all of CSS, a manipulation-based layout system for the web becomes possible, opening PBM to a very large audience.

ACKNOWLEDGMENTS

This work was supported in part by awards from the National Science Foundation (CCF–0916351, CCF–1139138, and CCF–1337415), as well as gifts from Google, Intel, Mozilla, Nokia, and Samsung.

REFERENCES

1. Badros, G. J., Borning, A., Marriott, K., and Stuckey, P. Constraint cascading style sheets for the web. UIST (1999), 73–82.
2. Badros, G. J., Borning, A., and Stuckey, P. J. The cassowary linear arithmetic constraint solving algorithm. CHI (2001), 267–306.
3. Barrett, C., Stump, A., and Tinelli, C. The Satisfiability Modulo Theories Library. `www.SMT-LIB.org`, 2010.
4. Bostock, M., Ogievetsky, V., and Heer, J. D3 data-driven documents. *IEEE Trans. on Visualization and Computer Graphics* (2011), 2301–2309.
5. Freeman-Benson, B. N. Converting an existing user interface to use constraints. In *UIST* (1993), 207–215.
6. Heer, J., and Bostock, M. Declarative language design for interactive visualization. *InfoVis* (2010), 1149–1156.
7. Heydon, A., and Nelson, G. The juno-2 constraint-based drawing editor. In *Digital Systems TR131a* (1994).
8. Hudson, S. E., and Yeatts, A. K. Smoothly integrating rule-based techniques into a direct manipulation interface builder. UIST (1991), 145–153.
9. Hurst, N., Li, W., and Marriott, K. Review of automatic document formatting. DocEng (2009), 99–108.
10. Jacobs, C., Li, W., Schrier, E., Bargeron, D., and Salesin, D. Adaptive grid-based document layout. *ACM Trans. Graph. 22*, 3 (2003), 838–847.
11. Lau, T. Why pbd systems fail: Lessons learned for usable ai. *AI Magazine* (2009), 65–67.
12. Lutteroth, C., Strandh, R., and Weber, G. Domain specific high-level constraints for user interface layout. *Constraints 13*, 3 (2008), 307–342.
13. Maloney, J. H. *Using constraints for user interface construction*. PhD thesis, Univ. of Washington, 1992.
14. McDaniel, R. G., and Myers, B. A. Getting more out of programming-by-demonstration. CHI (1999), 442–449.
15. Meyerovich, L. personal communication, 2012.
16. Moura, L. D., and Bjørner, N. Z3: An efficient smt solver. TACAS'08/ETAPS'08 (2008), 337–340.
17. Myers, B. A., and Buxton, W. Creating highly-interactive and graphical user interfaces by demonstration. SIGGRAPH (1986), 249–258.
18. Myers, B. A., Zanden, B. V., and Dannenberg, R. B. Creating graphical interactive application objects by demonstration. UIST (1989), 95–104.
19. Sannella, M. Skyblue: a multi-way local propagation constraint solver for user interface construction. UIST (1994), 137–146.
20. Singh, G., Kok, C. H., and Ngan, T. Y. Druid: A system for demonstrational rapid user interface development. UIST (1990), 167–177.
21. Sinha, N., and Karim, R. Compiling mockups to flexible uis. ESEC/FSE (2013), 312–322.
22. Viegas, F. B., Wattenberg, M., van Ham, F., Kriss, J., and McKeon, M. Manyeyes: A site for visualization at internet scale. *IEEE Trans. on Visualization and Computer Graphics* (2007), 1121–1128.
23. Vlissides, J. M., and Tang, S. A unidraw-based user interface builder. UIST (1991), 201–210.
24. Weitzman, L., and Wittenburg, K. Automatic presentation of multimedia documents using relational grammars. MULTIMEDIA (1994), 443–451.
25. Zeidler, C., Lutteroth, C., Sturzlinger, W., and Weber, G. The auckland layout editor: An improved gui layout specification process. UIST (2013), 343–352.
26. Zeidler, C., Lutteroth, C., and Weber, G. Constraint solving for beautiful user interfaces: How solving strategies support layout aesthetics. CHINZ (2012), 72–79.