Programming Layout by Manipulation

Thibaud Hottelier

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-158
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-158.html

August 18, 2014

Copyright © 2014, by the author(s).
All rights reserved.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to express my profound gratitude to my advisor Ras Bodik for his active participation and unwavering optimism, without which I would not have been able to see my research through. Thank you for your support and your patience.

I would like to thank my dissertation committee: Kimiko Ryokai for sharing her expertise in designing user studies; Paul Hilfinger for his insightful comments and challenging questions. I would also like to extend my appreciation to Per Ljung and Kimmo Kuusilinna, with whom I spent an exciting summer at Nokia Research.

Finally, I would like to thank Heather Levien for her invaluable help editing my papers as well as this dissertation.

Programming Layout by Manipulation

by

Thibaud Baptiste Hottelier

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Rastislav Bodík, Chair
Professor Paul Hilfinger
Professor Kimiko Ryokai

Fall 2014

Programming Layout by Manipulation

Copyright 2014
by
Thibaud Baptiste Hottelier

Abstract

Programming Layout by Manipulation

by

Thibaud Baptiste Hottelier

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodík, Chair

Creating layouts for documents, GUIs, or data visualizations is a time-consuming and error-prone process. Non-programmers would like the customization and fine-grained control currently only possible with handwritten layout engines. Today, such engines are written by proficient programmers. This thesis introduces new techniques for specifying layout and generating efficient layout engines.

First, we present a new programming methodology which addresses the two central sources of bugs in layouts: ambiguities and conflicts. Then, we introduce a language of constraints in which we capture layout specifications formally. Finally, we show how to generate efficient layout engines automatically. We provide the following individual contributions:

1. The Programming by Manipulation (PBM) paradigm targeted at non-programmers to establish specifications in visual domains such as layout. We introduce a new type of user demonstration—manipulation—which is resistant to users' imprecisions inherent in drawing. Instead of sketching the desired layout, users steer the exploration of potential layouts by pointing out what they would like to change.

2. L^3, a declarative language for layout specifications. L^3 is based on non-directional constraints in which the flow of computation is completely abstracted away.

3. A synthesis procedure—grammar modular synthesis—capable of generating layout engines from L^3 specifications. Our new algorithm scales to realistic layout specifications and produces generic engines supporting languages of documents.

To evaluate our work, we present two user studies showing not only that non-programmers can design interesting visualizations using PBM, but also that proficient programmers are more productive with PBM than with conventional constraint programming. We also compare the performance of our synthetized engines with state-of-the-art constraint solvers and show that our engines are up to two orders of magnitude faster.

Contents

List of Algorithms

List of Code Listings

List of Figures

List of Tables

Acknowledgments

I would like to express my profound gratitude to my advisor Ras Bodík for his active participation and unwavering optimism, without which I would not have been able to see my research through. Thank you for your support and your patience.

I would like to thank my dissertation committee: Kimiko Ryokai for sharing her expertise in designing user studies; Paul Hilfinger for his insightful comments and challenging questions. I would also like to extend my appreciation to Per Ljung and Kimmo Kuusilinna, with whom I spent an exciting summer at Nokia Research.

Finally, I would like to thank Heather Levien for her invaluable help editing my papers as well as this dissertation.

Chapter 1

Introduction

In practical terms, this thesis examines how to make layout programming easier and more accessible. By inferring specifications of visual layouts directly from user demonstrations and automatically synthesizing efficient layout engines, not only do we increase the productivity of programmers, but we also bring layout programming to a wider audience of non-technical users.

Visual layout is the art of arranging visual elements, such as paragraphs or images, in an aesthetically pleasing manner. Programming layout consists of establishing how the sizes and positions of each visual element are computed, either as a function of known values (*e.g.*, an image size) or of runtime inputs (*e.g.*, the screen size). The set of visual elements constitute a *document*; the program which computes the layout of a document is called a *layout engine*.

Both proficient programmers and non-technical users solve layout tasks, either by writing layout engine code directly (Bostock *et al.*, 2011) or by using tools such as WYSIWYG[1] editors (Viegas *et al.*, 2007). Tools accessible to everyone only allow limited customizability of layout; usually only the theme/visual appearance can be edited. Fine-grained control over algorithmic aspects, such as how positions are derived, requires manual modification of the layout engine, a task which many potential users cannot accomplish. Furthermore, writing a layout engine by hand is a significant undertaking, even for seasoned programmers. The turnaround between the conception of a layout design and the first layouts can take days. Prototyping is slow, thus making design tryouts expensive.

Layout engines are complex programs, not unlike compilers. When writing layout specifications, for instance by constraining positions of elements, programmers must carefully navigate between two hazards: *ambiguities* (under-specification) and *conflicts* (over-specification). Both bugs result in unexpected, sometimes non-deterministic layouts. The cascade of consequences leading to these bugs is difficult to track down and understand, even for experienced programmers. As a result, debugging is done by trial and error, a tedious process. Moreover, the

[1]What You See Is What You Get

performance of layout engines is critical. For instance, in interactive settings, the layout must be computed in a fraction of a second to appear responsive.

To address these problems, this dissertation proposes (i) a new programming methodology for layout, called Programming by Manipulation; (ii) a language of constraints by means of which layout specifications can be captured concisely; and (iii) a synthesis-assisted compiler producing efficient layout engines.

Programming by Manipulation (PBM) streamlines layout specification by allying user demonstrations with guided exploration of the layout design space. By inferring layout specifications from user demonstrations, Programming by Manipulation makes layout programming accessible to non-technical users.

We capture layout specifications formally in L^3, a declarative layout language based on constraint satisfaction. By casting layout as a satisfiability problem instead of the more common optimization formulation (Badros *et al.*, 1999; Schrier *et al.*, 2008), verifying static properties becomes tractable. As a result, PBM can prevent users from getting stuck in ambiguities or conflicts (contradictions). L^3 specifications are both concise and modular, thus facilitating code reuse.

Our compiler can generate layout engines automatically from L^3 specifications, making prototyping easy and inexpensive. The resulting engines are correct by construction and are as efficient as the engines manually written by expert programmers.

1.1 Dissertation Overview

The remainder of this chapter presents Programming by Manipulation informally, using examples to illustrate the three steps of layout creation: authoring, specification, and compilation. At each step, we discuss today's state-of-the-art techniques and present the rationale behind the major design decisions which led to PBM. The subsequent chapters of this thesis are organized as follows:

Chapter 2 We present Programming by Manipulation, a new programming methodology for authoring visual layout, targeted at non-programmers. Our approach addresses 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 manipulation tool indicates how the ambiguity can be removed, by computing how the elements just made free can be positioned or sized with available constraints.

We present the results of our user studies, demonstrating that both non-programmers and programmers can effectively use our prototype. Our results suggest that PBM is five times more productive than direct programming with constraints.

Chapter 3 We describe L^3 (Language for Layout Languages), our declarative constraint-based language for visual layout. L^3 is based on non-directional constraints and abstracts away the flow of computation. We explain how to specify new visual elements and create custom layout specifications of languages of documents. We introduce traits which bundle constraints in a modular and composable unit to promote code reuse. By restricting the legal nestings of visual elements, users can create languages of documents in L^3. We show that relational attribute grammars are suitable formalism to capture layout specifications.

Chapter 4 We present Grammar-Modular (GM) synthesis, an algorithm for program synthesis from large tree-structured relational specifications, such as the ones found in layout. GM synthesis makes synthesis applicable to previously intractable problems by decomposing them into smaller subproblems, which can be tackled in isolation using off-the-shelf synthesis procedures. The program fragments thus generated are subsequently composed to form a program satisfying the overall specification.

We apply GM synthesis to L^3 specifications and generate tailored layout engines for languages of documents. Our experimental results show that GM synthesis can successfully generate layout engines for non-trivial data visualizations, and that our synthesized engines are between 39- to 200-times faster than general-purpose constraint solvers.

1.2 Motivating Example

In this section, we explain the general steps in layout creation common to all techniques and tools. Visual layout is a vast domain which spans three principal applications: graphical user interfaces (GUIs), data visualizations, and documents. These three application domains have fuzzy boundaries. For instance, the layout of web pages is a hybrid between document and GUIs layouts. In this thesis, we focus on data visualization layouts, which offer the richest and most complex layouts of our three domains.

At a high level, 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.

More precisely, the visual elements to lay out are arranged in a hierarchical structure—a tree—called the *document*. The nodes of the document are either graphical entities or invisible positioning/grouping units. Each document node is decorated with attributes representing its sizes, positions, margins, colors, *etc.* Some attributes are marked as input. They are either known at design time (*e.g.*, the size of an image), or are runtime constants (*e.g.*, the size of the screen); all other attributes are unknown and must be computed by the layout engine by solving constraints. Ultimately, the document is passed to the renderer for display.

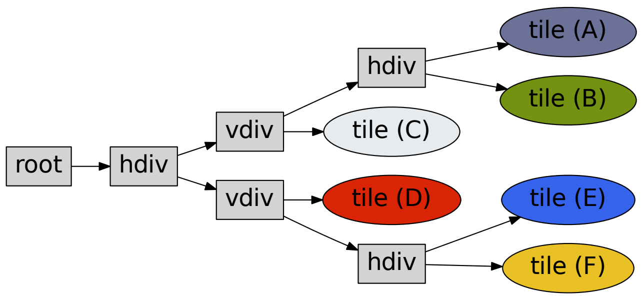
Layout Creation Independently from the techniques and tools used, the creation of new layouts can always be divided into the following three tasks: constructing a document from layout "bricks" called *blocks*; establishing the layout semantics of each block; and, finally, creating a layout engine supporting any document constructed from such blocks.

We illustrate each task by creating a treemap (Johnson and Shneiderman, 1991), a data-visualization based upon recursive tiling that depicts the relative sizes of objects. Treemaps are popular in finance to show the relative capitalizations of a group of companies[2]. We start from a dataset containing a list of companies, together with their capitalization. For our example, we use the small dataset constituted of six companies shown in Table 1.2.1.

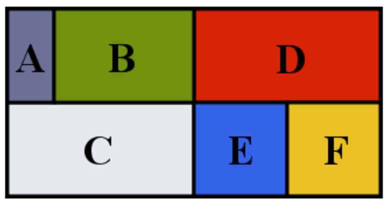
Name of Company	A	B	C	D	E	F
Capitalization	4	12	16	16	8	8

Table 1.2.1. A small dataset for our treemap. Each company is represented by a pair containing its name together with its capitalization.

Our goal is to create the visualization shown in Figure 1.2.1b. The treemap layout represents each company by a tile whose area is proportional to the company's capitalization.



(a) A document representing the dataset from Table 1.2.1



(b) The layout of the document

Figure 1.2.1. A treemap visualization. The tree of instances of blocks constituting a document (a), together with its layout (b). In the document, the hierarchy of dividers (*hdiv*, *vdiv*) encodes how the space is tiled.

The first step of the layout creation process consists of choosing a set of layout blocks. Then we construct a document representing the dataset by nesting the chosen blocks to form a tree. In essence, the document is a tree of nodes; nodes are instances of blocks. Informally, blocks are types/classes of nodes. Blocks define the layout semantics of nodes: typically, how each node's positions and sizes are computed. As such, blocks act as the types of document nodes.

[2]See SmartMoney's Map of the Market at www.marketwatch.com.

For the sake of the example, we are going to assume that we are given blocks with already correct layout semantics and that no further specification is required. In practice, we would have to either choose and fine-tune blocks from a library or create new ones ourselves. Choosing and specifying blocks is one of the crucial steps of layout creation. Each layout technique/tool has its own specific process.

For our treemap, we use the following four blocks: Document leaves represent companies and are instances of the *tile* block. Inner nodes encode how the space is tiled; they are either horizontal or vertical dividers (*hdiv* and *vdiv* blocks). Only *tiles* have a visual appearance; *h/vdivs* are invisible, they only position their children. Finally, the root of the document is a special *root* block. By nesting these blocks to form a document, we specify how the treemap is tiled. Figure 1.2.1a shows one document corresponding to our dataset. Many other tilings are possible.

Layout Computation Once the document is created, the next step consists of creating a layout engine, *i.e.* a program capable of computing the layout of the document. That is, computing values for all document attributes based on input attributes. Each valuation of document attributes is called a *layout*. For our treemap, the input attributes are the capitalizations of each of the companies. Once the layout is computed, the document is passed to the renderer which draws the document by reading the computed attribute values of each node.

Additionally, when the dataset changes, for instance, with new, up-to-date capitalizations, the layout engine can be run again on the same document (but with new values for the input attributes) to update the visualization. When we grow our dataset by adding new companies, we would like to reuse the same layout engine. As such, our layout engine must be generic enough to layout other documents constructed from the same set of blocks.

State-of-the-Art

Before describing how to specify block semantics to create a treemap by manipulation, we offer an overview of how this task would be accomplished without manipulation, using state-of-the-art tools and techniques. In the process, we outline the challenges faced today by layout designers.

Programmability How are layouts authored today? There are three main approaches for creating layouts such as a treemap, each offering a distinct trade-off between ease of use and how much control the designer has over the layout.

- *Canned Layouts* At one end of the spectrum, the most accessible approach consists of picking a "canned" layout from a layout library such as Protovis or D3 (Heer and Bostock, 2010; Bostock *et al.*, 2011). Such libraries provide ready-to-use layout blocks; no programming skills are necessary. However, the customization of the layout is limited to selecting a visual "theme". Visual aspects such as colors can be modified graphically by non-programmers, but the more fundamental and algorithmic aspects of the layout are fixed. To see why this is insufficient, imagine a biologist creating a phylogenetic tree representing the evolutionary branching of species. The layering of nodes has an important biological meaning. As such, precise control over this aspect of the layout is crucial to him. This problem is particularly acute for scientific visualizations, which have very specific layout requirements.

Moreover, layout libraries offer only a limited number of layout designs to choose from. However, the universe of designs is virtually infinite. By considering only positional aspects of tree layouts, such as the overall architecture (flat, radial, flower, *etc.*, see Figure 2.2.3), the layering and space allocation strategies, we count over a hundred plausible tree designs. It is unlikely that our biologist will readily find the required tree layout in a library.

- *Handwritten Custom Engines* At the other end of the spectrum, writing the layout engine by hand offers the most flexibility. However, this approach requires advanced technical expertise. As such, it is inaccessible to a large audience of potential users, such as scientists, most of whom do not program. Furthermore, layout engines are complex programs; writing them by hand is time consuming. Based upon our experience, this task takes on the order of days, even for seasoned programmers. As a result, trying out design ideas is expensive.

- *Constraints* Constraints offer a middle ground between canned layouts and handwritten layout engines. Constraints are arguably the most widespread and successful layout programming technique. For instance, the foundations of TEX are laid upon constraint. CSS, the ubiquitous web template language, also relies on constraints, although in a more restricted and indirect manner[3]. With canned layouts, designers express their intent indirectly, by composing a document from available blocks. By stating high-level properties of the layout directly (*e.g.*, element A is left aligned with element B), constraints promise to yield precise and predictable layouts.

  However, programming with constraints is error-prone. Bugs in constraints are manifested by either ambiguities or conflicts (*i.e.*, inconsistencies). Ambiguities arise when not enough constraints are stated, resulting in documents which can be laid out in multiple, distinct, and often unexpected ways. In practice, the layout chosen by the engine is unlikely to be the indented one. Conflicts are caused by contradictory constraints; fulfilling all constraints becomes impossible. Consequently, to produce some layout, some constraints are disregarded by the layout engine, once again causing unexpected layouts. Either way, both ambiguities and conflicts make the resulting layout difficult to anticipate; it appears to the programmer as if chosen arbitrarily.

[3]CSS constraints are limited to unary functions of the parent's attributes (Hurst *et al.*, 2009).

Each of the three approaches has significant drawbacks. Creating complex layouts such as data visualizations remains a difficult task for non-technical users and a time-consuming and error-prone one for programmers today.

Performance Computing layout quickly enough is also a major technical challenge. For instance, today's web pages are typically constituted of over a thousand document nodes, for each of which dozens of attributes must be computed in a fraction of a second to provide a responsive user experience in interactive settings. Furthermore, data visualizations tend to be much larger than web pages.

When layout is expressed with constraints, the layout engine can be implemented by invoking a *general-purpose constraint solver* such as Cassowary (Badros *et al.*, 2001a). Each solution to the constraints is a distinct layout. In practice, for performance reasons, all web browsers and most visualization libraries rely on *specialized solvers*, manually tailored to solve one particular type of layout. Such handcrafted solvers achieve efficiency by fixing a static traversal schedule of the document. For instance, all CSS engines compute width before height. However, such specialized engines restrict the type of constraint supported, thus limiting the range of expressible layouts.

As an example of the rigidity caused by fixing the computation strategy, assume you would like to layout a document with a sidebar which must be wide enough to display all of its content on a single screen but must not overflow past the bottom of the screen. The contents of the main panel, however, are allowed to overflow. To compute such a layout, one would first compute the width of the sidebar, given the screen height, and then compute the main area height given the sidebar width. This design is impossible to implement with CSS, which must always compute height as a function of width.

To conclude, writing layout engines for constraint-based layout is challenging. The performance of general-purpose constraint solvers is not satisfactory for the strict performance requirements of layout, and handwritten layout engines place restrictions on layout computations which restrain designers.

Design Principles

We synthesize the challenges posed by the current layout programming techniques into four desirable properties, covering both programmability and performance. These properties form the design principles of Programming by Manipulation.

1. *Accessibility* Non-technical users should be able to control and customize the algorithmic aspects of the layout such as, for instance, how the positions of an element are computed.

2. *Predictability* Layout should be predictable. That is, the layout specification must be deterministic: there must exist a unique layout satisfying the specification. The programmer should be able to build a mental model of the computation taking place in the layout

engine. He should be able to understand how the resulting layout is entailed by his specification.

3. *Flexibility* The creation of layouts should not be hindered by computational artifacts imposed by the layout engine such as a fixed computation strategy (Figure 3.1.2). The layout engine must adapt to the layout specification, and not vice-versa.

4. *Efficiency* Layout engines should be fast enough to be part of the interactive loop which responds to user interactions.

From the four design principles outlined above, we derive three technical challenges. This thesis proposes solutions for each of them:

- How to capture the layout specification at a level of discourse understandable by non-technical users.
- How to alleviate and, if possible, entirely prevent ambiguities and contradictions, which we believe are the two principal sources of bugs in layout specifications.
- How to generate efficient layout engines automatically from layout specifications.

We first outline how our approach addresses each of these technical challenges. Then, in the following three subsections, we describe our solutions informally, using examples, and present our design rationale.

Our Approach With PBM, designers specify layouts using special demonstrations called manipulations. Using our manipulation tool (the *manipulator*), designers correct an existing layout. Our tool infers the behaviors of blocks from the manipulations performed.

The manipulator prevents users from creating conflicts. Ambiguities are explained with a visual summary highlighting which aspects of the layout are constrained and which aspects are still “free”. The manipulator exploits this freedom to propose alternative layouts, facilitating the exploration of potential layout designs. At the end of the manipulation process, we obtain a specification of the layout blocks, based on constraints, which is free of both conflicts and ambiguities.

Finally, by leveraging program synthesis, we compile the layout specification obtained by manipulation into an efficient layout engine. Our synthesized layout engines are expressed as a set of traversals over the document tree, like the tailored solvers handcrafted today by expert programmers.

1.3 Programming by Manipulation

We illustrate the Programming by Manipulation workflow by creating a phylogenetic tree. Phylogenetic trees represent the evolutionary branching of species over time (Figure 1.3.1). We proceed in two main steps. First, we create a sample document by nesting blocks drawn from a library of blocks. PBM blocks are flexible: they bundle many alternative constraints, for instance one per alignment strategy. In a second step, using the manipulator, we "browse" potential layouts of the sample document. Each layout stems from selecting a distinct combination of alternative constraints in blocks. Each combination is called a *configuration*. We steer the exploration by directly interacting with the layout of the sample document. By choosing one configuration, the manipulator selects constraints for each block, thus establish their layout semantics. The result a set of *configured* blocks in which all alternative constraints have either been made mandatory or removed. Configured blocks constitute a layout specification.

Walkthrough To construct a phylogenetic tree, we choose the following three blocks from the PBM library: *treeRoot*, *innerNode*, and *treeLeaf*. By selecting blocks, we define the broad class of tree layouts, for instance whether the layout is Cartesian or Polar. In our case, we select the three blocks forming the base of all Cartesian tree layouts. PBM blocks were created by an expert programmers using L^3 constraints.

Our blocks include alternative constraints for each design aspect (*e.g.*, placement of tree nodes into layers, spacing between layers, spacing between siblings, *etc.*). For instance, to layer tree nodes, our library includes the following three alternative constraints: (L_1) the same layer for all nodes; (L_2) layering based on tree depth; or (L_3) layering based on a node's distance to its furthest leaf. Within each layer, other alternative constraints permit siblings to be placed equidistant to each others, proportionally to their size, or proportionally to the size of their subtree. The goal of manipulation is to "configure" each flexible block with a definite set of constraints.

In the second step, we construct a sample document using our three flexible blocks. The purpose of the sample document is to provide a support on which we will demonstrate the desired layout. As such, we do not need to encode a large dataset into the sample document. A small but representative subset will suffice. To avoid confusion between the representation of the document as a tree of nodes and its layout, which is also a tree, we show the sample document in XML.

```
<treeRoot>
  <innerNode><treeLeaf/><treeLeaf/></innerNode>
  <innerNode>
    <innerNode><treeLeaf/><treeLeaf/></innerNode>
    <treeLeaf/>
  </innerNode>
```

```
</treeRoot>
```

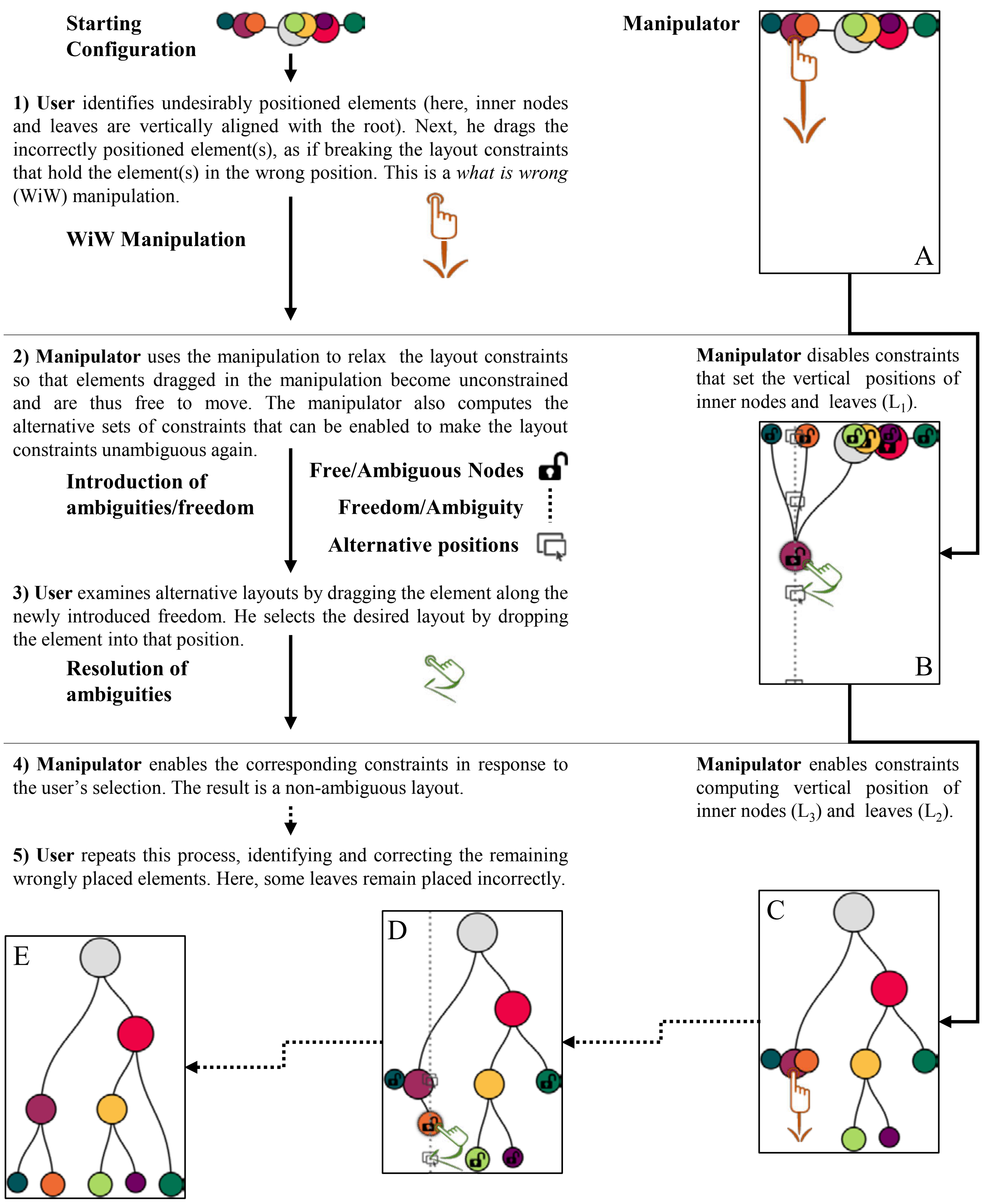
Listing 1.3.1. The sample document. We are going to demonstrate the desired layout using this document. It is constructed from the three flexible tree blocks drawn from the PBM library.

Recall that blocks are flexible in that they bundle alternative constraints. The manipulator starts by selecting an arbitrary set of alternative constraints (*i.e.*, an arbitrary configuration), solves them, and displays the resulting layout. Of course, this layout is unlikely to be the desired one; we correct it by manipulation. At a high-level, PBM proceeds as follow: First, we highlight one undesirable aspect of the layout by displacing one or more nodes. This constitutes a *what is wrong* (WiW) manipulation. The manipulator responds by removing spurious constraints, thus introducing ambiguities in the layout. The manipulator also proposes how these ambiguities can be resolved. Each resolution corresponds to adding new constraints. We choose one resolution and repeat this process until we achieve the desired layout.

We illustrates this process in detail on Figure 1.3.1. We find the initial configuration undesirable: inner nodes and leaves have been placed in the same layer (L_1). To correct the layout, we drag one inner node downward (WiW manipulation), asking the manipulator to remove constraints L_1, thereby introducing freedom to move some elements (these are the new ambiguities). This moves the system to configuration B. Unlocked icons indicate partially constrained nodes. These are (ambiguous) nodes that are free to move. A dashed line explains that the selected node can move vertically. The manipulator proposes alternative options to "fix" these free nodes; each is represented by an icon. For each such icon, some constraints are enabled. When we drag the selected node and hovers it above one such icon, the manipulator shows a preview of the resulting layout. Dropping the node into one such position adds the selected constraints (L_3 for inner nodes, L_2 for leaves) leading to configuration C. C has no ambiguity. We can now proceed to identify the next layout error (layering of leaves) and repeat this process, moving from C to E.

Using Ambiguities to Drive Exploration Unfortunately, the manipulator cannot prevent both ambiguities and conflicts while letting users browse configurations freely. When removing or adding constraints through manipulations, we will be confronted by one or the other, depending on which operation is performed first. PBM rules out conflicts while explaining ambiguities with visual summaries.

PBM prevents conflicts from occurring by proposing to add constraints only if they are conflict-free. When blocks are fully constrained (there is no freedom/ambiguity left), PBM forces users to first remove constraints using a WiW manipulation before they can add new ones. The manipulator explains ambiguities visually, using dashed-lines to summarize the freedom that is present in the layout. For instance, in layout B of Figure 1.3.1, the purple node is vertically free. That is, its vertical coordinate is unconstrained. As such, it could be laid out anywhere along the dashed-line, subject to available constraints. The manipulator proposes alternative positions



1) User identifies undesirably positioned elements (here, inner nodes and leaves are vertically aligned with the root). Next, he drags the incorrectly positioned element(s), as if breaking the layout constraints that hold the element(s) in the wrong position. This is a *what is wrong* (WiW) manipulation.

2) Manipulator uses the manipulation to relax the layout constraints so that elements dragged in the manipulation become unconstrained and are thus free to move. The manipulator also computes the alternative sets of constraints that can be enabled to make the layout constraints unambiguous again.

Manipulator disables constraints that set the vertical positions of inner nodes and leaves (L_1).

3) User examines alternative layouts by dragging the element along the newly introduced freedom. He selects the desired layout by dropping the element into that position.

4) Manipulator enables the corresponding constraints in response to the user's selection. The result is a non-ambiguous layout.

Manipulator enables constraints computing vertical position of inner nodes (L_3) and leaves (L_2).

5) User repeats this process, identifying and correcting the remaining wrongly placed elements. Here, some leaves remain placed incorrectly.

Figure 1.3.1. Demonstrating the node layering of our phylogenetic tree. The hand icon illustrates the manipulation performed by the user. In configurations A and B, links between tree nodes are too short to be visible.

for the purple node, enabling us to browse candidate layouts by alternatively introducing and removing ambiguities in the design. By explaining ambiguities at a level understandable by non-technical users, PBM turns ambiguities into a mechanism for exploring the space of layout designs.

Interestingly, recent work took the opposite approach. ALE (Zeidler *et al.*, 2013), a state-of-the-art graphical user interface builder, rules out ambiguities by design and explain conflicts by computing the maximum satisfiable set of constraints. We believe that, compared to conflicts, ambiguities are more amenable to be conveyed with summaries understandable by non-programmers. To explain a conflict, one must show why something is impossible, a property intrinsically difficult to visualize.

1.4 A Language of Constraints for Layout

To demonstrate the layout of our phylogenetic tree, we constructed a sample document by instantiating flexible blocks. Blocks are created by an expert programmer using L^3, a declarative language of constraints. Constraints are packaged in composable modules called traits. L^3 supports definitions of small "domain-specific" languages of documents using tree grammars.

Blocks Recall that a document is a tree of nodes. A block is akin to the type of a document node and determines its layout semantics. Blocks define a set of attributes (*e.g.*, positions and sizes) as well as constraints over these attributes. To promote code reuse, blocks are constructed by composing small bundles of constraints called *traits*. Within blocks, some constraints are optional; others are mandatory. Optional constraints provide flexibility, non-programmers can add or remove them using the manipulator. Mandatory constraints capture essential features such as the definition of coordinate systems. All constraints are local; they refer to attributes of the direct parent and children in the document hierarchy.

To illustrate the features of L^3, we show two traits, both used in the definition of the *hdiv* block of the treemap.

```
mandatory trait RelCartesian {
    top + height = bot
    left + width = right
    x = parent.x + left
    y = parent.y + top
}

optional trait HAlignChild0 {
    child0.left = 0
    child0.right = width
    child0.right = child1.left
    child0.left = child1.left
    child0.right = 0
```

```
        child0.left = width
        ...
    }

    block hdiv with RelCartesian, HAlignChild0, ...
```

Listing 1.4.1. Two traits: one mandatory, one containing optional constraints. Both are composed (together with other traits) to define the *hdiv* block. Optional constraints will be configured by manipulation. Equal symbols denote equality, not assignment. Qualifiers *parent* and *childN* are used to refer to attributes of the parent and of the N^{th} child, respectively. Attributes *left* and *top* are the horizontal and vertical displacements of a node with respect to its parent.

The first trait (*RelCartesian*) sets up a relative coordinate system based on the built-in (absolute) Cartesian coordinates. As shown in Figure 1.4.1, the origin is in the top left corner. With the relative coordinative system, each box can position itself relative to its parent via the *left*, *top* attributes. These attributes control the horizontal/vertical displacement of a node (green box) with respect to the top-left corner of its parent (blue box). All constraints of *RelCartesian* are mandatory.

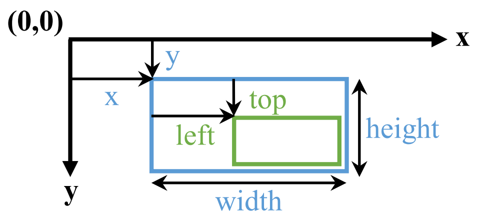


Figure 1.4.1. The Cartesian relative coordinate system. Using attributes *left* and *top* defined by *RelCartesian*, the green box can be positioned relative to its parent, the blue box. Blue attributes belong to the blue box; green attributes belong to the green box.

The second trait (*HAlignChild0*) enumerates horizontal alignment strategies. These constraints are optional; a subset of them will be selected by manipulation. For instance, the first constraint (*child0.left* $=$ 0) set the left displacement of the first child of a node to zero. As a result, the child and its parent (green and blue boxes in Figure 1.4.1) are left-aligned. For this particular trait, in most cases, only a single constraint will be selected at any given time.

Non-Directionality One key feature of L^3 constraints is their non-directionality: they abstract away the flow of computation. For instance, the constraint stating that two children are left justified (*child0.left* $=$ *child1.left*) does not specify whether the first child is positioned in function of the second child or vice versa. We delegate the determination of the flow of computation to the compiler, thus raising the level of abstraction. The non-directionality of L^3 constraints has the following two benefits:

- Non-directionality enables the same block to be reused across layouts requiring distinct flows of computations. This feature is particularly useful in interactive layouts, where values may flow in either direction depending on the user's action. Imagine a scroll-box with a slider indicating the position of the viewport (Figure 1.4.2). The respective positions of the slider and the content displayed in the viewport are bound by the following relationship:

$$\frac{slider\_position}{slider\_height} = \frac{content\_position}{content\_height}$$

  When the user drags the slider, the position of the content must be updated by the layout engine. Conversely, when the user grows the content, for instance by typing text, the position of the slider must be recomputed. In either case, the same constraint is maintained. With directional constraints, we would have to create two scroll-box blocks, one for each flow of computation. Non-directionality lets us capture both scenarios concisely, with the same L^3 constraint, thus promoting code reuse. Furthermore, from the same specification, we can automatically generate two layout engines, each updating the scroll-box layout in response to one of the two user interactions.

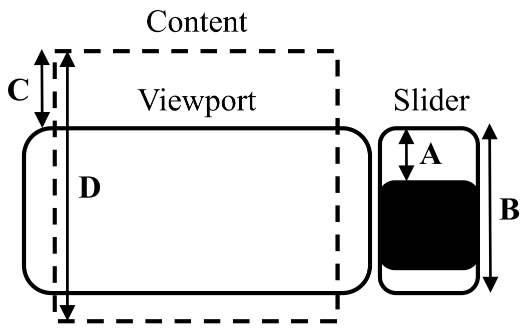


Figure 1.4.2. A diagram of a scroll-box. Labels denote the slider position (A), the slider height (B), the content position (C), and the content height (D).

- Non-directional constraints greatly simplify the implementation of the manipulator by abstracting away functional dependencies (*i.e.*, the set of attributes read and computed by each constraint). Values must be available for all attributes read by a constraint before it can be used to compute more attributes. For instance, with directional constraints, the manipulator would have to ensure the absence of cyclic dependencies when adding or removing optional constraints. By delegating this task to the L^3 compiler, the manipulator can focus on interpreting users' manipulations.

Languages of Documents Datasets inevitably change and grow. Therefore, layout specifications must be general enough to apply to updated documents reflecting such changes. In L^3, programmers can define languages of documents by specifying which block nestings are legal. For our treemap visualization (Figure 1.2.1), we can define a language of treemap documents by restricting block nestings as follows: (i) leaves of the document must be instance of the *tile* block; (ii) inner document nodes must be instances of either *hdiv* or *vdiv*; (iii) *hdiv* and *vdiv* nodes must alternate; and (iv) the root of the document is an instance of the *root* block. Languages of documents are specified with regular tree grammars. We show below the grammar for our language of treemaps.

```
S ::= root(H)
H ::= hdiv(V, V)        V ::= vdiv(H, H)
   |  tile()               |  tile()
```

Listing 1.4.2. The tree grammar of our treemap language. Non-terminals are capital letters, terminals are blocks. Notice how the grammar enforces the alternation of *hdiv* and *vdiv* blocks

Note that the document shown in Figure 1.2.1a belongs to our treemap language. Given a language of documents, our L^3 compiler is capable of generating layout engines supporting any document belonging to the language.

1.5 Grammar-Modular Synthesis

To generate layout engines automatically from L^3 specifications, we introduce a new program synthesis algorithm called *grammar-modular* synthesis. Program synthesis (Manna and Waldinger, 1980; Pnueli and Rosner, 1989) is the task of transforming high-level specifications (*e.g.*, relations) into executable programs (*e.g.*, functions). Before we outline our algorithm, let us take a step back and discuss the possible implementations of layout engines. There are two kinds of layout engines for constraint-based layouts:

- *General-Purpose Constraint Solvers* Each document defines implicitly a constraint system over its attributes as the conjunction of all block constraints. As such, given a document, a general-purpose constraint solver can compute its layout (Sannella, 1994; Badros *et al.*, 2001a). Such solvers perform a potentially expensive backtracking search to find a solution to the constraints; no analysis is performed off-line. In general, these engines support a broad class of constraints.

- *Tree Traversals* Layout engines expressed as a series of traversals over the document tree can solve the layout in linear time. Their efficiency stems from a static traversal schedule; *i.e.* they do not determine dynamically which attributes must be evaluated next. These engines are usually implemented manually by expert programmers for a specific, restricted class of constraints. In practice, for performance reasons, all web browsers and most visualization libraries use tree-traversal layout engines.

Instead of solving L^3 constraints with a general-purpose constraint solver, we leverage program synthesis to generate tree-traversal layout engines tailored to a language of documents expressed in L^3. In essence, synthesis automates the optimizations currently performed by the expert programmers who write tree-traversal layout engines for visualization libraries.

There are two technical challenges preventing us from using state-of-the-art synthesis techniques such as Sketch or Comfusy (Solar-Lezama *et al.*, 2006; Kuncak *et al.*, 2010):

- *Scalability* Layout specifications are significantly larger than what state-of-the-art synthesizers can currently handle. For instance, the average web page contains over one thousand document nodes, each with dozen of attributes. Current synthesis techniques scale up to approximately 100 program variables.
- *Genericity* Program synthesis techniques generate one program from one relational specification. In layout, each document constitutes a layout specification. As a result, regardless of scalability, such techniques generate layout engines supporting only a single document. When the document is modified, for instance by adding more companies and subdivisions in a treemap, the layout specification implicit in the document also changes. As a result, a new layout engine must be synthesized. To be practically useful, our layout engines must be generic enough to handle languages of documents. Thus we need a synthesis procedure capable of handling specifications of languages of documents.

Grammar-modular (GM) synthesis builds on top of state-of-the-art synthesis techniques to meet the requirements outlined above. The key benefit of our synthesis is shifting the cost of the backtracking search performed by general-purpose constraint solvers to compilation time, leaving only value propagations and function applications for layout time. Our algorithm takes advantage of the tree structure present in documents to decompose the specification into smaller subproblems. Each subproblem can then be tackled in isolation with standard techniques and their results are combined to form a layout engine.

Grammar-Modular Synthesis To synthesize a layout engine, we first decompose the L^3 specification of a layout language at the level of blocks. Each block becomes an independent synthesis problem. Then we use existing synthesis techniques to generate as many *local functions* as possible for each block. Each local function computes some of attributes of its block in terms of other attributes from the same block. In essence, we functionalize the non-directional L^3 constraints; each local function represents one possible flow of computation. The final step consists of choosing some local functions in each block and compose them together to create a layout engine. This is the crucial step of GM synthesis. The resulting layout engine is an attribute grammar (Knuth, 1968). That is, the execution of the local functions is syntax-directed by the structure of the document to layout. As such, the resulting layout engine is capable of computing the layout of any document constructed from the same set of blocks. Ultimately, the attribute grammar can be scheduled into efficient tree traversals.

Example We illustrate GM synthesis on our treemap example. The goal is to create a layout engine for our language of treemaps (Listing 1.4.2). The specification of a language of documents is a set of configured blocks together with a grammar of legal nestings. Recall that by configuring blocks with the manipulator, we selected a subset of alternative constraints which are now mandatory; all non-selected constraints were removed. We show below all the constraints defining the *hdiv* block after configuration by manipulation.

```
configured block hdiv {
    top + height = bot
    left + w = right
    x = parent.x + left
    y = parent.y + top
    parent.scale = scale
    scale * cap = height * width
    height = child0.height = child1.height
    children.top = 0
    child0.left = 0
    child0.right = child1.left
    cap = child0.cap + child1.cap
}
```

Listing 1.5.1. The constraints defining the *hdiv* block. At this point, the *hdiv* block has been configured by manipulation; all constraints are mandatory.

By applying synthesis locally on the *hdiv* block alone, we functionalize L^3 constraints into local functions. Each local function computes some attributes of *hdiv* in terms of other *hdiv* attributes. For instance, the constraint $scale * cap = height * width$ yields the following four local functions:

$$scale := \frac{height * width}{cap} \qquad cap := \frac{height * width}{scale}$$

$$height := \frac{scale * cap}{width} \qquad width := \frac{scale * cap}{height}$$

The first local function can compute the scale if the height, width, and capitalization are known. Each local function performs one possible propagation of values among the four attributes bound by the constraint. Informally, we refer to such propagations of values as alternative flows of computation.

The next step consists of determining which of these four functions must be used in the layout engine. To do so, we now reason globally, on the entire language of treemaps. The goal is to select just enough local functions within each block to be able to compute all attributes. Choosing a sufficient set of local functions is the crux of GM synthesis. By doing so, we determine the flow of computation through each block. In fact, the set of selected local functions can be thought of as the semantic/evaluation rules of an attribute grammar.

We show below the local functions selected by GM synthesis for the *hdiv* block:

$$\begin{aligned}
bot &:= (height + top) & right &:= (width + left) \\
x &:= (left + parent.x) & y &:= (top + parent.y) \\
scale &:= parent.scale & cap &:= (child0.cap + child1.cap) \\
child0.height &:= height & child1.height &:= height \\
child0.top &:= 0.0 & child1.opt &:= 0.0 \\
child0.left &:= 0.0 & child1.left &:= child0.right \\
height &:= \frac{scale * cap}{width} & &
\end{aligned}$$

At this point, we have constructed a layout engine capable of solving any document in our language of treemap (Listing 1.4.2).

Finally, an attribute grammar scheduler can compile the set of selected local functions into tree traversals. For instance, in our example, the *cap* attributes are computed first with a bottom-up traversal, then the *scale* attributes are computed top-down. The final layout engine for our language of treemaps is constituted of five tree traversals.

Our compiler uses the Superconductor (Meyerovich *et al.*, 2013) attribute grammar scheduler which can produce parallel and incremental layout engines. Ultimately, our layout engines are compiled down to JavaScript and can be easily deployed in any web browser. We show below the final product of the compilation: the five visitors of the *hdiv* block.

```
hdiv.prototype.visit0 = function() {
    this.getChild(0).left = 0.0;
    this.getChild(0).top = 0.0;
    this.getChild(1).top = 0.0;
}

hdiv.prototype.visit1 = function() {
    this.cap = this.getChild(0).cap +
               this.getChild(1).cap;
}

hdiv.prototype.visit2 = function() {
    this.right = this.width + this.left;
    this.scale = this.parent_scale;
    this.height = this.scale * this.cap / this.width;
    this.getChild(0).parent_scale = this.scale;
    this.getChild(0).height = this.height;
    this.getChild(1).parent_scale = this.scale;
    this.getChild(1).height = this.height;
}

hdiv.prototype.visit3 = function() {
    this.getChild(1).left = this.getChild(0).right;
}

hdiv.prototype.visit4 = function() {
    this.bot = this.height + this.top;
    this.x = this.parent_x + this.left;
    this.y = this.parent_y + this.top;
    this.getChild(0).parent_x = this.x;
    this.getChild(0).parent_y = this.y;
    this.getChild(1).parent_x = this.x;
    this.getChild(1).parent_y = this.y;
}
```

Listing 1.5.2. The five visitors of the *hdiv* block implemented in JavaScript. Visitors 0, 2, and 4 are executed in top-down traversals; visitors 1 and 3 are executed bottom-up.

Using grammar-modular synthesis, we compiled the L^3 specification of our treemap language (itself configured by manipulation) into an efficient layout engine.

1.6 Collaborators and Publications

The work described in thesis is the fruit of a collaboration with Rastislav Bodík, James Ide, Doug Kimelman, Per Ljung, and Kimiko Ryokai and has been introduced in prior publications (Hottelier *et al.*, 2014; Hottelier and Bodik, 2014).

Chapter 2

Programming by Manipulation

This chapter introduces a new programming methodology targeted at non-programmers for specifying document layout—Programming by Manipulation (PBM). Users demonstrates the desired layout on a sample document, directly interacting with its concrete layout by means of special demonstrations called manipulations. The layout specifications are captured formally in a language of constraints called L^3 (Chapter 3). Such specifications can ultimately be compiled to an executable layout engine working not only on the sample document used to demonstrate the layout, but also on other documents constructed from the same blocks (Chapter 4).

We start by introducing the principles behind PBM: how we exploit ambiguities to navigate the space of possible layout designs and how we prevent users from entering conflicting requirements (Section 2.1). The following two sections (Sections 2.2 and 2.3) focus on the programming methodology and present PBM from a user's point of view. Then, we formalize the concepts behind PBM and describe the central component (the PBM manipulator) as well as its implementation (Section 2.4). Finally, we present our evaluation of the effectiveness of PBM and report the results of two user studies on both proficient programmers and non-programmers (Section 2.5).

2.1 Motivation and Approach

Today, the task of building complex layouts such as data visualizations requires the advanced technical expertise of trained programmers. Libraries of "prepackaged" layouts offer only limited design options to non-programmers. As as result, custom visualizations are out of reach of many potential users, such as scientists, most of whom do not program. Before presenting our solution, we summarize the programmability challenges posed by constraints.

Programming with Constraints

By stating properties of the layout directly, constraints promise to yield a precise, high-level, 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 (Section 2.5) 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 (Badros *et al.*, 2001a; Zeidler *et al.*, 2012).

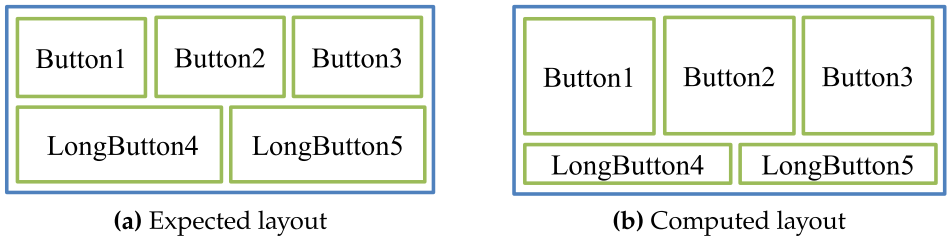
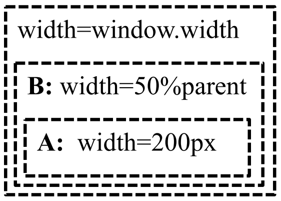


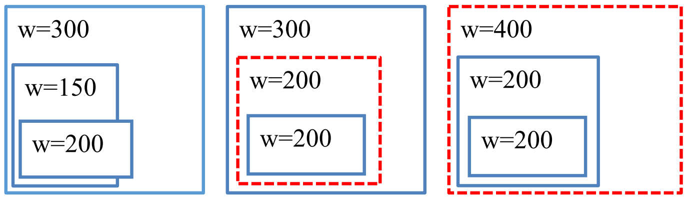
Figure 2.1.1. Unexpected spring-effects. On a document with 2 rows of buttons, we give the same preferred height (*i.e.*, the same spring force) to all buttons. We expect layout (a) but get layout (b) instead. The combined spring force of the three buttons of the top row is stronger than that of the two buttons of the second row, effectively squeezing them vertically. Example from Zeidler *et al.* (2012).

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 instance, it is well known that optimization-induced "spring-effects" result in unexpected layouts (Zeidler *et al.*, 2012), forcing designers to twiddle with constant parameters by trial and error (Figure 2.1.1). 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.

For example, with CSS, text overflowing the borders of a container is a classic illustration of conflict resolution not matching the designer's intent. This phenomenon occurs with only two boxes: Box A containing the text with a preferred width of 200px, and its decoration, box B, set to half as wide as the window (Figure 2.1.2a). 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 300px, CSS will overflow the text of A out of B (Figure 2.1.2b). If B has a visible border, the resulting layout is unlikely to please.



(a) A conflicting document



(b) Three layouts, each results from dropping one constraint (from left to right): the containment constraint (A contains B), A's width constraint, and the window's width constraint.

Figure 2.1.2. Dropping constraints leads to unpredictable layouts. When the window width is 300px, the document is conflicting (a). To display the document, solvers drop some constraints, creating many alternative ways to lay out the document. Three of which are shown (b).

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. 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 illustrate this problem with an example inspired by ALE (Lutteroth *et al.*, 2008; Zeidler *et al.*, 2013). 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.

We conclude that the fundamental 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 (Zeidler *et al.*, 2013). 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.

Design Principles

With the understanding that ambiguities and conflicts are the central programmability issues to address, let us now look at which other properties are desirable to build a practical system for specifying layouts by demonstration. In particular, we look at the needs of data visualization which is arguably the most challenging of our 3 layout domains (beside document and GUI).

Data visualization spans across the entire spectrum of layout types, from flow layouts to guillotine layouts. Contrary to GUI layouts, data visualizations can be non-boxy (a radial tree) or recursive (a treemap). As such, our approach cannot be tailored to any particular types of layouts.

Datasets, which users turn into documents, 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.

Finally, as with all visual domains, user demonstrations of layouts are never pixel-perfect; they contain small imprecisions which should not derail the PBD process.

We summarize our design principles in the following four points:

1. Ambiguities and conflicts must either be ruled out or be explained at a level of discourse understandable by non-programmers.

2. The system must be resistant to the small imprecisions present in drawing-based demonstrations.

3. The language of constraints must be rich enough to capture a wide class of data visualizations, including recursive and non-boxy ones.

4. Users must be able to demonstrate the desired layout on a small subset of their data. The demonstration must generalize to other datasets.

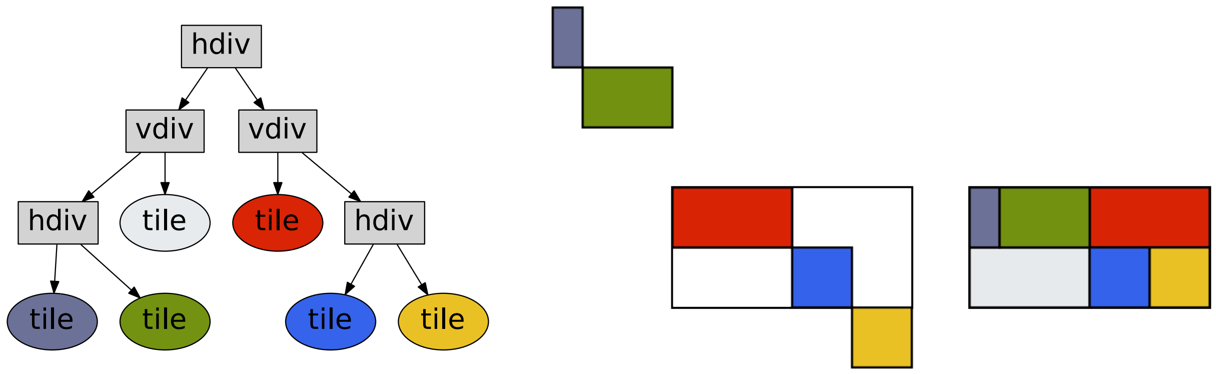
Programming by Manipulation

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 (Zeidler *et al.*, 2013), which rules out ambiguities and explains conflicts.

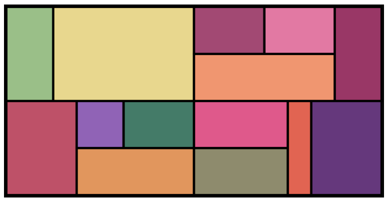
Creating a data-visualization with PBM proceeds in two main stages (Figure 2.1.3a). First, the user selects the broad class of the layout, *e.g.*, a tree, by building a sample document. This document is constructed by instantiating building blocks from a library. Each block is flexible, in that it encapsulates 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, he steers the exploration by manipulating the layout of the sample document by dragging blocks.

Our tool—the PBM manipulator—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 2.1.4a). Like StopThat interactions (McDaniel and Myers, 1999), 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, the manipulator only considers the "direction" of the WiW manipulation as opposed to the final locations of the displaced blocks. Behind the scenes, the manipulator determines which constraints to toggle off to allow the block to move in the demonstrated direction (Figure 2.1.4b). 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 2.1.4c). Only safe specializations (conflict-free) are proposed.

Once the layout has been configured, we know which constraints to use: we have established



(a) First the user create a sample document, a tree of blocks (left). Then, starting from an arbitrary initial configuration (middle), the user 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).

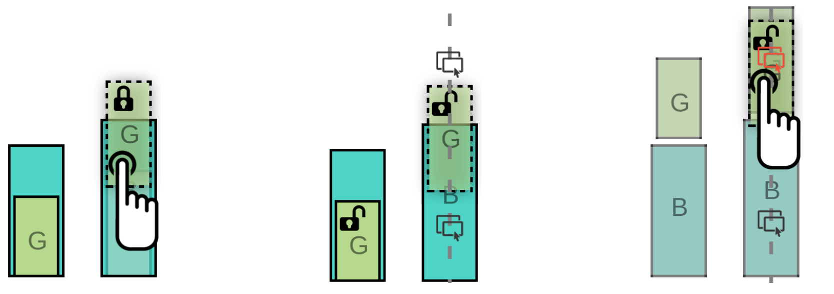
Figure 2.1.3. Creating a treemap with PBM. The 3 stages (a) to establish a layout specification, which is reusable (b).

the complete specification of the layout. We can finally turn the configuration into a layout engine which takes the document as a runtime input and lays it out (Figure 2.1.3b).

Design Rationale

In the next two paragraphs, we take a step back to explain the design rationale behind the following two features of PBM: (i) why exploration is a crucial feature of our approach; and (ii) why we choose to prevent conflicts and explain ambiguities, and not vice-versa.

Let's start by noting that the space of layouts established by the sample document is too large to be explored exhaustively: the total number of combinations of switches grows exponentially with the number of optional constraints. As such, we need an effective way to converge quickly on interesting layouts. Our early prototypes asked users to sketch the desired layout by repositioning all layout elements in one big demonstration. They performed rather poorly: the combination of switches inferred was rarely producing the desired layout, leaving users perplexed and without knowing neither what they did wrong nor how to improve their demonstration. The reasons for this failure are twofold: (i) drawing imprecisions create interferences; (ii) users are unaware of which layouts are expressive from the constraints embedded in the sample document. While the first point is specific to our domain, visual layout, the second point is a fundamental flaw of



(a) WiW Manipulation **(b)** Generalization **(c)** Specialization

Figure 2.1.4. 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).

traditional programming by demonstration: Lau eloquently points out in "Why PBD Fails" (Lau, 2009) that opaque design spaces whose boundaries are not discoverable by users is one of the main reason PBD systems have not been as successful as expected. The target program, as represented in a user's mind, is often not expressible. Interestingly, Lau notes that there often exists an equivalent program or a close approximation which is expressible and for which users would settle. This led us to our exploration-centric approach which enables users to discover by themselves good enough layouts.

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 alleviate 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.

2.2 Overview of Programming by Manipulation

This section provides a detailed overview of layout by manipulation, using a phylogenetic tree as a running example (Figure 2.2.2). The goal for our user is to establish the core aspects of layout, such as position, size, alignment, and margins.

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 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. We call each block instance a (document) *node*. By choosing which blocks to use, the layout 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 *innerNode*, and the leaves are *treeLeaf*. Figure 2.2.1 shows one possible sample document for our phylogenetic tree. 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. We discuss how to specify "families" of documents in Section 3.2.

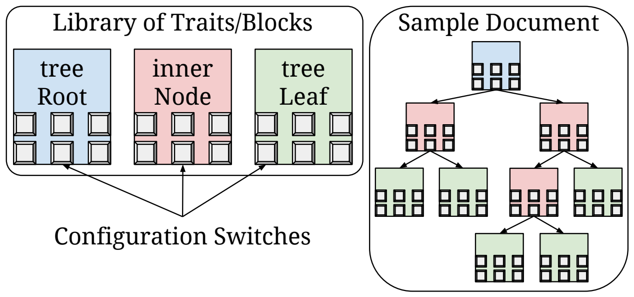


Figure 2.2.1. A sample document for a tree layout. The user chooses three blocks in the library: *treeRoot*, *innerNode*, and *treeLeaf*. Each block defines configuration switches controlling which constraints are enabled. The user creates a sample document by nesting instances of blocks.

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 layout 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 introduce the language constraints behind blocks in Section 2.4 and detail it further in Chapter 3.

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:

- A document is *deterministic* if it admits exactly one layout.
- A document which admits more than one layout is *ambiguous*.
- A document for which there exists no layout is inconsistent: some of its constraints are *conflicting*.

We compute the layout of a configured document by solving the corresponding constraint system. Modern solvers such as Z3 (De Moura and Bjørner, 2008) 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.

A *layout engine* is an executable layout configuration: a program which takes as input any document built from the same set of blocks and computes a layout for it. Some templates can be compiled into efficient tree-traversal layout engines. Unlike general purpose constraint solvers, such engines do not perform a costly backtracking search. We discuss how to compile configurations to layout engines in Chapter 4.

Demonstrating the Layout Configuration

To establish the finer aspects of the visualization, the layout 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, devised to help layout designers finding an interesting layout quickly, even in huge configuration spaces. PBM turns conventional demonstrations upside down: Instead of directly demonstrating the goal, layout designers highlight one layout aspect (*e.g.*, horizontal alignment) they would like to change by dragging one document node away from its constrained position. We call such manipulations "what is wrong" (WiW) manipulations.

The manipulator presents the layout of the sample document according to the currently active configuration. The exploration always starts from an arbitrary configuration that yields a deterministic document. Then, by manipulating the layout itself, the layout 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 layout designer either (i) points out an incorrect layout feature; or (ii) chooses an alternative layout from among a range of options. Figure 2.2.2 details the exploration steps and Figure 1.3.1 illustrates the user interactions, both on our phylogenetic tree. In four manipulations, we establish the desired tree layering.

The layout 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[1] 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 user highlights incorrect aspects of the layout by dragging nodes to displace them from the position constrained by the current configuration. Generalizations are triggered by the "Break Rules" button (Figure 2.2.3a).

- *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, users choose one layout from a list of alternatives.

To browse configurations effectively, layout 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, the manipulator must also convey which constraints can be added

[1]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.

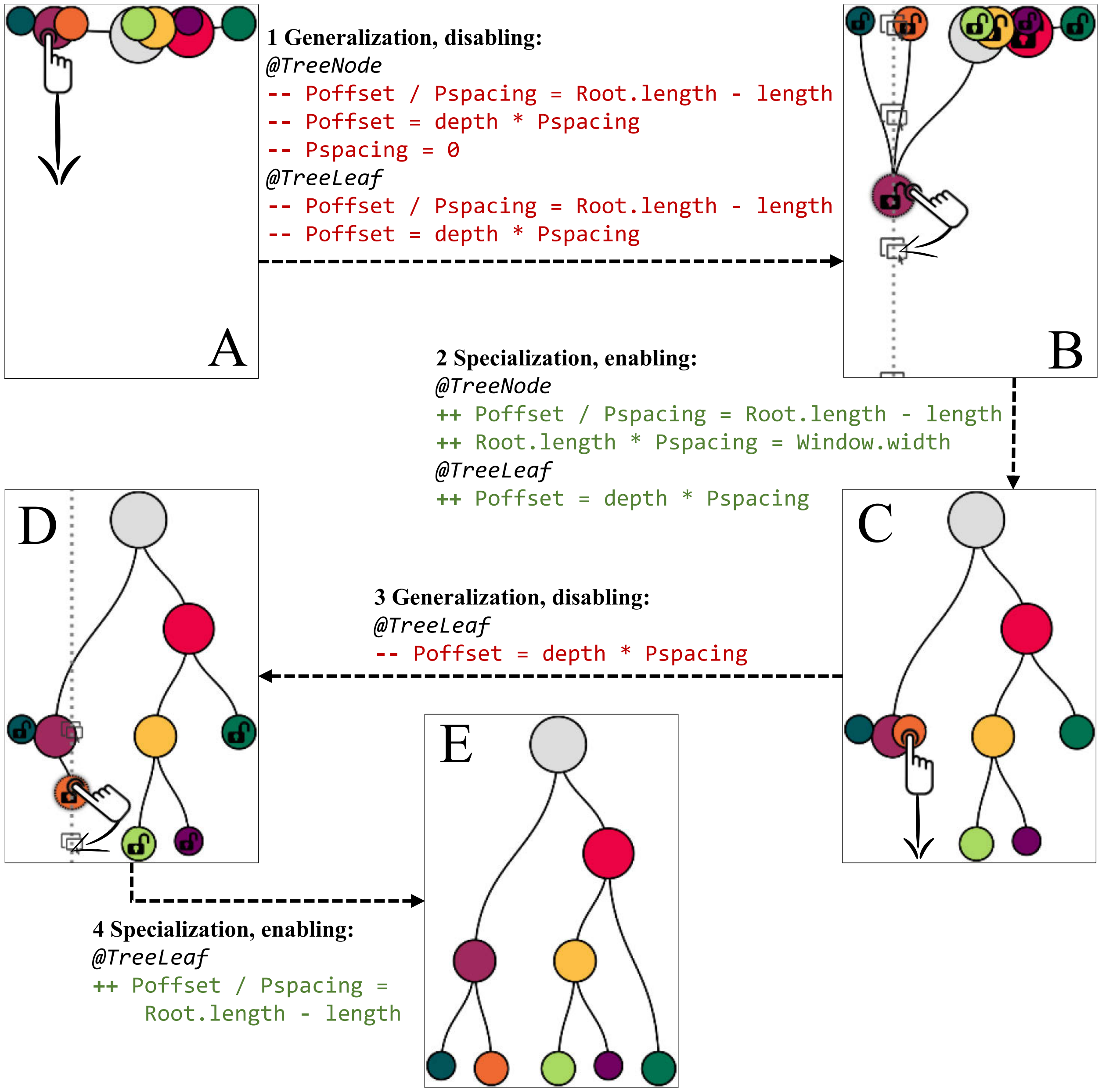
safely to specialize the current configuration. This translates into two responsibilities for the manipulator: explaining ambiguities and proposing potential resolutions.

Introducing Ambiguities by Breaking Constraints To control the introduction and resolution of ambiguities, users need to know not only which nodes are free (*i.e.*, not fully constrained) but also which aspects (*e.g.*, height or horizontal position) of the nodes 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 2.1.4, 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 users at a glance.

We visualize the ambiguity of a layout with dashed lines, which show which nodes 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 users, we display one admissible layout, augmented with axes of freedom, one per dimension of the ambiguity base. The manipulator highlights free nodes with unlocked icons. When the user selects a free node, the manipulator 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, the manipulator understands the visual semantics of common block attributes. Figures 2.2.3a and 2.2.3b 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 (Hudson and Yeatts, 1991). For instance, nodes which are dragged beyond the layouts admissible by current configuration should resists the user's action by snapping back to a legal position. As a result, users can "feel" constraints by manipulating nodes. This also enables us to explain generalizations with the following UI metaphor: by dragging a node past its snapping radius, the user is"breaking" constraints. We detail how to compute ambiguity bases in Section 2.4.

Proposing Resolutions (Specializations) After the designer breaks some constraints, he 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



1 Generalization, disabling:

```
@TreeNode
-- Poffset / Pspacing = Root.length - length
-- Poffset = depth * Pspacing
-- Pspacing = 0
@TreeLeaf
-- Poffset / Pspacing = Root.length - length
-- Poffset = depth * Pspacing
```

2 Specialization, enabling:

```
@TreeNode
++ Poffset / Pspacing = Root.length - length
++ Root.length * Pspacing = Window.width
@TreeLeaf
++ Poffset = depth * Pspacing
```

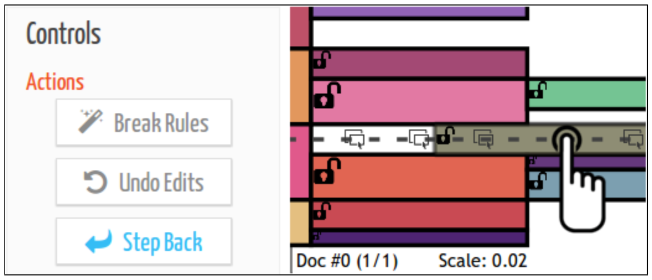
3 Generalization, disabling:

```
@TreeLeaf
-- Poffset = depth * Pspacing
```

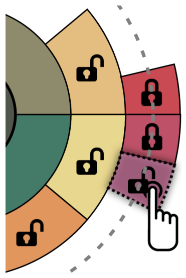
4 Specialization, enabling:

```
@TreeLeaf
++ Poffset / Pspacing =
     Root.length - length
```

Figure 2.2.2. The five configurations explored to specify 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. To illustrate the effects of generalizations and specializations, we give the constraints disabled (red) and enabled (green) by each manipulation. The constraints shown control the computation of *Poffset*, the distance between a node and its parent. For each node, the *depth* attribute is the distance to the root node; the *length* attribute is the distance to its farthest leaf.



(a) A flat ambiguity base and the manipulator 's GUI



(b) Radial

Figure 2.2.3. Two examples of ambiguity bases. Ambiguous (free) nodes are indicated with unlocked icons. The manipulator displays the ambiguity base of the selected node with axes of freedom (dashed-lines). Here, the bases of each selected node are constituted of a single axis. To enable users to "feel" constraints, each node moves freely along its axes of freedom, but resists displacement in other directions by snapping to its axes. We also show the manipulator's user interface (a). The "Break Rules" button triggers generalizations.

To communicate available specializations to the layout designer, we represent each of them as a point in the ambiguity space. Given a free node, we mark the points on its axes of freedom where the node would be positioned if that specialization were chosen. Our interface uses semantic snapping to emphasize these specialization points (Hudson and Yeatts, 1991). The user chooses one of them by drag&dropping a free node onto one of its specialization points. For example, in Figure 2.1.4b, 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 user drags and drops one green bar onto a marked site, as illustrated in Figure 2.1.4c.

2.3 Creating New Traits From Constraints

In this section, we briefly explain how the expert programmer creates new traits. Each trait bundles constraints pertaining to one layout aspect such has horizontal alignment. The expert programmer maintains a library of traits in which the user can choose and compose traits to form new blocks. To give the flavor of L^3, we present two examples of traits. For a detailed description of L^3, we refer the reader to Chapter 3.

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.* Under the hood, our language of constraints is based on SMT theories (Barrett *et al.*, 2010) which provides an expressive set of primitive constructs.

We present below one trait setting up a polar coordinate system. It is used by all our radial layouts (*e.g.*, Figure 2.2.3b).

```
mandatory trait Polar2Cartesian {
    x = radius * cos(angle)
    y = radius * sin(angle)
}
```

While most constraints are enabled by configuration switches, there exists a few mandatory ones for essential features such as coordinate systems, as shown above. We show below a trait of optional constraints for controlling the horizontal alignment of the first child of a box. The `optional` keyword indicate that not all constraints must be upheld; the user will choose a subset of them by manipulation. Note that the configuration switches are implicit. Equal symbols signify equality, not assignment.

```
optional trait HAlignChild0 {
    child0.left  = 0
    child0.right = width
    child0.right = child1.left
    child0.left  = child1.right
    child0.right = 0
    child0.left  = width
    ...
}
```

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.

2.4 The PBM Manipulator

In this section, we present the foundations behind the PBM manipulator, the principal component of PBM. The manipulator interprets user manipulations and translates them into either generalizations or specializations. We start by enumerating the tasks which must be accomplished by the manipulator. Then, we formalize key concepts such as generalization and specialization which were introduced in the previous two sections (Sections 2.2 and 2.3). Finally, we explain how the manipulator fulfills each of its tasks and outline our implementation.

The manipulator has to function out-of-the-box with any blocks written by the expert programmer, and do so without requiring any modification. In essence, the manipulator must be constraint-agnostic. The responsibilities of the manipulator can be summarized in five computational tasks. Given a sample document, the manipulator must:

1. Find a deterministic configuration. This is configuration is the starting point of the user-guided exploration.

2. Given a configuration, compute the layout of the sample document.

3. Given a configuration, compute its ambiguity base.

4. Given configuration, compute which specializations are safe (*i.e.*, conflict-free).

5. Given a WiW manipulation, generalize the current configuration.

We accomplish these five tasks by encoding the entire configuration space defined by the sample document in SMT theories. Conveniently, our satisfiability approach to layout lets us cast all five manipulator tasks as satisfiability queries. Before detailing the computation of each manipulator task, let us introduce the formal bases on which we will define generalizations, specializations, and finally ambiguity bases.

Preliminaries

We start by stating formally concepts which were previously introduced informally in Section 2.2. Recall that a block is defined by a set of constraints, some of which can be activated or disabled by configuration switches. Blocks are detailed in Section 3.2.

Definition 1 (Document)**.** *A* document *is a tree of block-labeled nodes. For every document d, let $rel(d)$ be the underlying constraint system representing d.*

Formally, $rel(d)$ is constructed by taking the conjunction of the constraints behind each node in d. The variables of $rel(d)$ are either (i) document attributes; or (ii) boolean configuration switches.

Definition 2 (Configuration)**.** *A* configuration *of a document is a boolean valuation of its configuration switches. We write d_c to denote the document d configured with configuration c.*

Let $rel(d_c)$ be the constraints system resulting from applying configuration c to document d. Formally, $rel(d_c)$ is constructed by substituting each configuration switch of $rel(d)$ by its value in c. As such, the only variables remaining in $rel(d_c)$ are document attributes.

We define two convenience functions—$on(c)$ and $off(c)$—which return, respectively, the set of enabled and disabled switches of configuration c.

Definition 3 (Layout)**.** *A* layout *of document d under configuration c is a solution of $rel(d_c)$. A configuration is* conflicting *if $rel(d_c)$ admits no layout,* deterministic *if $rel(d_c)$ admits exactly one layout, and* ambiguous *if $rel(d_c)$ admits two or more layouts. Let a* safe *configuration be either deterministic or ambiguous.*

In practical terms, a layout is a mapping from document attributes to values. We write $x.a$ to refer attribute a from the node x. Given a layout l, $l[x.a]$ denotes the value of attribute a from x in layout l.

The space of all possible configurations of a document has a lattice structure. Let C be the set of all configurations for a given document. We define the partial order $\sqsubseteq$ on C as follows:

$$\forall c_1, c_2 \in C. \quad c_1 \sqsubseteq c_2 \Leftrightarrow on(c_1) \subseteq on(c_2).$$

As such, $L = (C, \sqsubseteq)$ forms a complete subset lattice. The lattice order is connected with the solutions of $rel(d_c)$, as shown below:

Lemma 1. *Configurations and layouts of a given document obey the following correspondence:*

$$\forall c_1, c_2 \in C. \quad c_1 \sqsubseteq c_2 \Leftrightarrow rel(d_{c_1}) \supseteq rel(d_{c_2}).$$

Consequently, we can also view the order $\sqsubseteq$ as a measure of determinism. Greater configurations are more deterministic thus less ambiguous. The infimum (bottom element) of L is the configuration with the most ambiguity: none of its constraints are enabled. Conversely, to supremum (top element) of L is the configuration with all constraints enabled. This configuration is likely to be conflicting.

Recall that the manipulator prevents users from encountering conflicting configurations. As such, users only explore a subset of the configurations of C, the safe ones. To formally define generalizations and specializations—the two moves users can make through the configuration space—we introduce a new lattice containing only safe configurations. Let $L_s = (C_s, \sqsubseteq)$ be the lattice representing the space of explorable configurations, where $C_s \subseteq C$ is the set of safe configurations. Since we have removed conflicting configurations, L_s is only a lower semi-lattice.

We are now ready to define generalizations and specializations on L_s. Generalizations can be viewed as moving down one level in L_s. Conversely, specializations are defined as moving up in L_s by one level.

Definition 4 (Generalization/Specialization)**.** *Let $gen(c) = max(lb(c) \setminus c)$ and $spec(c) = min(ub(c) \setminus c)$ denote, respectively, the set of* generalizations *and* specializations *of configuration c, where lb/up denotes the set of lower/upper bounds in L_s and max/min denotes the set of maximal/minimal elements in C_s with respect to $\sqsubseteq$.*

Intuitively, $gen(c)$ are the configurations immediately below c in the lattice L_s. Conversely, $spec(c)$ are the configurations immediately above c. Note that $spec(c) = \varnothing$ for all deterministic configurations. Interestingly, the process of navigating the space of configurations by manipulation is not monotonic: by alternating generalizations and specializations, the user moves both up and down in L_s.

Finding an Initial Configuration and Computing Layout

To accomplish the five computational tasks of the manipulator, we encode both $rel(d)$ and $rel(d_c)$ in SMT theories (Barrett *et al.*, 2010). This enables us to phrase each of of the five manipulator tasks in terms of satisfiability queries. As a result, we can leverage the power of SMT solvers to perform heavy computations. Our implementation of the manipulator uses Z3 (De Moura and Bjørner, 2008) as solver. We briefly outline our encoding before explaining how to compute the manipulator tasks.

We encode $rel(d)$ by prefixing optional constraints with boolean guards acting as configuration switches:

```
guard1 ⇒ optional_cstrnt1 ∧ mandatory_cstrnt ∧
guard2 ⇒ optional_cstrnt2 ∧ ...
```

Each individual constraint is further encoded in the appropriate SMT theory. To produce parametric layout engines capable of working on multiple documents built out of the same set of blocks, we make the assumption that all nodes of the same block type obey the same set of constraints. We enforce this property by sharing the same guard variables across all nodes labeled with the same block.

Initial Configuration The first computational task consists of finding an initial configuration which deterministic. While it would be possible to exhaustively compute all safe configurations (C_s) by enumerating the solutions of $rel(d)$, such an enumeration would be impractical since C_s can be very large (exponential in the number of optional constraints). Furthermore, most configurations would likely never be visited by users, effectively wasting computations. To avoid a costly enumeration, we opted for a greedy approach: we query the solver for more deterministic (more switches toggled on) configurations until we obtain a deterministic one, as illustrated in Algorithm 2.4.1. Once an initial deterministic configuration has been found, all computations are performed on-demand, in response to users' actions.

There exists documents for which Algorithm 2.4.1 will not find a deterministic configuration. Such cases arise in when an ambiguous configuration c has no upper-bound in L_s other than itself. In other words, when an ambiguous configuration which cannot be made deterministic by enabling one or more of its currently disabled constraints; doing so would always result in a

Algorithm 2.4.1. Finding an initial configuration which is deterministic. Note that solutions to $rel(d)$ provides us not only with a safe configuration c but also with one possible layout l of d_c.

```
Input: A sample document $d$
Output: A deterministic configuration of $d$
begin
    /* Start with an arbitrary configuration $c$ and its layout $l$. */
    $(c, l) \leftarrow \text{SatModel}(rel(d))$
    while $IsSat(rel(d_c) \wedge \neg l)$ do /* Check if $c$ is ambiguous. */
        $S_{on} \leftarrow \bigwedge_{s \in on(c)} s = \top$
        $S_{more} \leftarrow \bigvee_{s \in off(c)} s = \top$
        /* Find a new configuration with at least one more switch
           enabled. Such a configuration is guaranteed to exist only
           in well-formed documents. */
        $(c, l) \leftarrow \text{SatModel}(rel(d) \wedge S_{on} \wedge S_{more})$
    end
    return $c$
end
```

conflicting configuration. In such cases, Algorithm 2.4.1 fails when it queries the solver for a safe configuration more deterministic than the current one.

We define a class of well-formed documents on which Algorithm 2.4.1 always succeeds.

Definition 5 (Well-Formed Documents)**.** *A document d is* well-formed *iff all ambiguous configurations have at least one upper bound in L_s other than itself. That is, $\forall c \in C_s.\ ub(c) \supset c$. Or equivalently, iff for every ambiguous configuration c of d there exists at least one configuration switch s in $off(c)$ such that $rel(d_c) \wedge s = \top$ is satisfiable.*

Lemma 2. *Algorithm 2.4.1 always successfully find a deterministic configuration on well-formed documents.*

We consider that it is the responsibility of the expert programmer to ensure that documents are well-formed when creating traits. Note that malformed documents pose problems beyond just computing the initial configuration. When navigating the space of configurations, malformed documents create situations in which the user faces a configuration with ambiguities but cannot resolve them because there exists no safe specialization. He must first introduce more ambiguities (*i.e.*, generalize) to reach a configuration which can be specialized. Such situations are counter-intuitive thus undesirable.

Computing Layout The second computational task, computing the layout of d under configuration c, can be performed simply by querying a satisfiable assignment to the document attributes of $rel(d_c)$. To do so, we encode to constraint system of a configured document, $rel(d_c)$ by replacing each boolean guard in $rel(d)$ by its value in c.

Computing the Ambiguity Base

For the third computational task, the manipulator computes the ambiguity base of a configuration. Recall that the ambiguity base is needed to augment layouts shown to users with axes of freedoms which visually convey which ambiguities are present in the current configuration. Informally, an ambiguity base is a summary of all the layouts admissible by a document d under a given configuration c.

More formally, an ambiguity base is an *algebraic base* of the solution space of $rel(d_c)$. If document d has n attributes, each ambiguity base of d (one per configuration) spans a subspace of $\mathbb{R}^n$. The ambiguity base of deterministic configurations has zero dimension; it is a single point in $\mathbb{R}^n$. The base of ambiguous configurations has one or more dimensions.

Definition 6 (Ambiguity Base)**.** *Given a document d and a configuration c, the* ambiguity base *of c is an algebraic base of the solutions of $rel(d_c)$. Each point in the ambiguity base is a layout of d under configuration c.*

Unfortunately, computing ambiguity bases for all but the smallest documents is intractable in practice. If all constraints in d are linear equalities, Gaussian elimination can compute ambiguity bases efficiently. But in practice, we observed that most layouts contain non-linear constraints and sometimes also inequalities. For such constraints, the best known algorithm to compute ambiguity bases is Cylindrical Algebraic Decomposition (CAD) (Collins, 1975), which has a worst-case complexity triply exponential in the number of document attributes. Even small documents with a dozen boxes can have over hundred attributes. As such, using CAD to compute ambiguity bases would be prohibitively expensive, especially since the manipulator must be fast enough to be used interactively. Consequently, our implementation of the manipulator relies on an approximation of ambiguity bases which is fast to compute.

Given a configuration c, we approximate its ambiguity base by computing the subset of document attributes which can admit more than one than one value in d_c. Each such attribute becomes one independent dimension of the approximated ambiguity base. As such, our approximation ignores all relationships between attributes forming the ambiguity base. For instance, the size of an image might be unconstrained with the exception of its aspect ratio: $4 * height = 3 * width$. Such relationships are lost by our approximation: we will consider both *height* and *width* to be independent dimensions. Algorithm 2.4.2 describes how to compute approximated ambiguity bases.

Definition 7 (Approximate Ambiguity Base)**.** *Given a document d and a configuration c, let M be the subset of attributes of d which admit multiple values in the solutions of $rel(d_c)$. The canonical* *(or natural) base stemming from turning each attribute in M into a unit vector is an* approximated ambiguity base.

In essence, we over-approximate the precise ambiguity base with an enveloping base of higher dimensionality. In practice, we found that our approximated bases were good enough for explaining to users which ambiguities are present in a given configuration.

Lemma 3. *Approximated ambiguity bases are sound over-approximations. The approximated ambiguity base of a configuration c spans the entire subspace defined by the precise ambiguity base of c. Consequently, for all configurations, the dimensionality of the approximated ambiguity base is always larger or equal to that of the precise ambiguity base.*

Algorithm 2.4.2. Computing the ambiguity base of a configuration. Recall that a document is a tree of nodes labeled with blocks. Notice that we do not need to iterate over all document attributes since all nodes of the same block type share the same constraints. Consequently, we only iterate over all attributes once per block, as opposed to once per node.

```
Input: A sample document $d$ and a configuration $c$
Output: An approximated ambiguity base of $c$

begin
    $B \leftarrow \varnothing$
    /* Find one layout of $d_c$. */
    $l \leftarrow \text{SatModel}(rel(d_c))$
    foreach box $b$ in $d$ do
        foreach attribute $a$ of box $b$ do
            /* Lookup value of $a$ for any instance of $b$ in $l$. */
            Let $x$ be any node labeled with block $b$ in $d$.
            $v \leftarrow l[x.a]$
            /* Check if $a$ can have a value other than $v$. */
            if $IsSat(rel(d_c) \wedge (x.a \neq v))$ then
                $B \leftarrow B \cup \{b.a\}$
            end
        end
    end
    return $B$
end
```

Computing Generalizations and Specializations

It remains to explain the last two computational tasks of the manipulator: (i) computing the set of safe specializations of the current configuration; and (ii) generalizing a configuration given a WiW manipulation. We start with computing safe specializations.

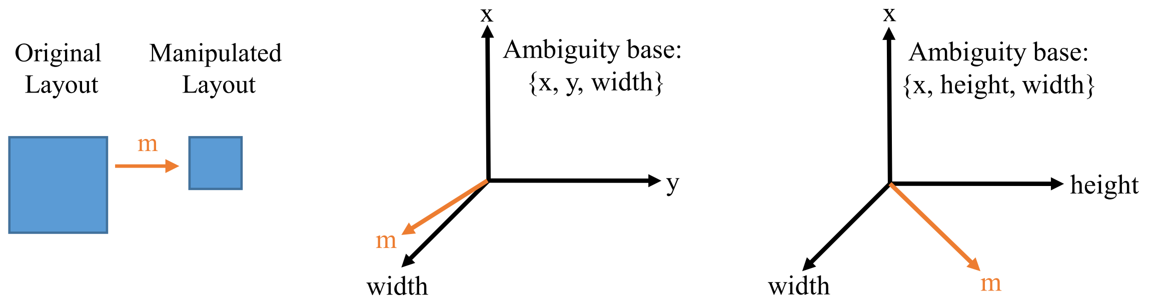
Formally, for any configuration c, we must compute $spec(c)$, the set of safe specializations over L_s. We can do so with a straightforward iterative process (Algorithm 2.4.3). For each disabled configuration switch s in c, we enable s and attempt to compute a layout for the resulting configuration. We reject all configurations which do not admit any layout and thus are conflicting.

Algorithm 2.4.3. Computing the set of safe specializations of a configuration. We denote by $c[s := \top]$ the configuration resulting from enabling switch s in c

```
Input: A sample document $d$ and a configuration $c$
Output: The set of safe specializations $spec(c)$
begin
    $R \leftarrow \varnothing$
    foreach switch $s$ in $off(c)$ do
        if $IsSat(rel(d_c) \wedge s = \top)$ then
            $R \leftarrow R \cup \{c[s := \top]\}$
        end
    end
    return $R$
end
```

Unlike specializations, generalizations are always safe: conflicts cannot be introduced by disabling constraints. However, contrary to specializations which are chosen explicitly by the user when he drops a element onto a specialization point, generalizations are selected indirectly with WiW manipulations. To generalize the current configuration, the manipulator must decide which constraints to disable (or equivalently which ambiguities to introduce) based on the user's WiW manipulation.

First, we compute $gen(c)$, the set of candidate generalizations of the current configuration c. Since generalizing is always safe, this is set is simply an enumeration of all configurations with one less switch toggled on than c. Our task is to choose the "best" candidate with respect to the WiW manipulation. Intuitively, the best candidate is the configuration whose ambiguities are most in line with the WiW manipulation. For the sake of explanation, let's assume that only one block was displaced by the user. Consequently, we abstract the WiW manipulation into one vector, capturing the direction of displacement. Then, we rank candidates by computing a score based on how many dimensions of their ambiguity base align with the manipulation vector.


Original Layout
Manipulated Layout
m
Ambiguity base: {x, y, width}
x
y
width
m
Ambiguity base: {x, height, width}
x
height
width
m

(a) Manipulation **(b)** Candidate 1 **(c)** Candidate 2

Figure 2.4.1. Generalizing a configuration given a WiW manipulation. On a document consisting of a single box, the user performs the following WiW manipulation: shrinking the box in half, effectively dividing both its height and width by 2 (a). This manipulation is represented by the vector $\vec{m}$. We consider two generalization candidates and show their respective ambiguity base (b&c). The manipulation vector aligns with one dimension (`width`) of the first candidate's base. However, the same manipulation vector aligns with two dimensions (`height`, `width`) of the second ambiguity base. Consequently, we would prefer candidate 2 over candidate 1. Informally, the best candidate is the one whose ambiguities express $\vec{m}$ most completely.

Informally, we are counting how many of the principal components of the manipulation vector can be expressed in terms of the ambiguity base, as illustrated in Figure 2.4.1.

Definition 8 (WiW Manipulation)**.** *Given a document d, a* WiW manipulation *is a vector $\vec{m}$ in $\mathbb{R}^n$, where n is the number of attributes of d, capturing the changes to the layout of d made by the user.*

More technically, we compute the dot-product between the manipulation vector $\vec{m}$ and each dimension of the ambiguity base, both normalized to unit length. We consider that $\vec{m}$ aligns with an ambiguous dimension if their dot-product is greater than some threshold. We chose $\frac{1}{\sqrt{k}}$, where k is the dimensionality of the ambiguity base. This threshold corresponds to the value of the components of a unit vector of dimension k whose components are all equal. In practice, we found that this threshold seems to work well. To rank generalization candidates, we assign to them a score equal to the number of dimensions which align with $\vec{m}$. Algorithm 2.4.4 details the score computation.

Once the scores for all generalization candidates have been computed, the manipulator chooses the candidate with the highest score. Note that our ranking method does not impose a total order on candidates, but we found ties to be a rare occurrence in practice. As a last resort, we break ties by also considering the positions of the displaced elements and selecting the configuration admitting the closest layout with respect to a sum of squared differences metric. Let l_m be the layout of the document d after being manipulated by the user. We break ties by

Algorithm 2.4.4. Ranking generalization candidates. The score represents how well the ambiguities of the candidate align with the WiW manipulation performed by the user.

```
Input: A generalization candidate $c$ and a WiW manipulation $\vec{m}$.
Output: A score capturing how well the ambiguities of $c$ align with $\vec{m}$.
begin
    $r \leftarrow 0$
    $\vec{m}_0 \leftarrow \frac{\vec{m}}{\|\vec{m}\|}$
    $B \leftarrow \text{AmbiguityBase}(c)$
    $k \leftarrow \text{rank}(B)$
    foreach dimension $\vec{d}$ of $B$ do
        // We assume that $\vec{d}$ is already unit length.
        if $\vec{d} \cdot \vec{m}_0 \geq \frac{1}{\sqrt{k}}$ then
            $r \leftarrow r + 1$
        end
    end
    return $r$
end
```

choosing the configuration c which minimizes

$$\delta(l_m, c) := \min_{l \in rel(d_c)} \left(\sum_{\substack{\text{node} \\ x \in d}} \sum_{\substack{\text{attribute} \\ a \in x}} (l_c[x.a] - l_m[x.a])^2 \right).$$

To compute $\delta(l_m, c)$ without having to enumerate all layouts admitted by d_c, we perform a binary search on the minimum value of $\delta(l_m, c)$ for any layout of d_c.

2.5 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 the manipulator to design data visualizations?

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

To investigate these two questions, we conducted two user studies. In the first, we asked non-programmers to configure five data visualizations using the manipulator. 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 the manipulator and with an interface mimicking standard constraint programming.

Non-Programmers

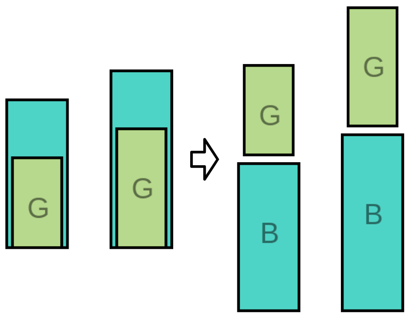
We recruited 11 participants (3 males, 8 females, ages 22 to 39) either students or staff from outside the engineering disciplines, largely from the Biology and Linguistics departments. Participants were selected for their lack of formal training in programming. When shown a picture of an icicle graph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

Experimental Setup Each session proceeded as follows: Participants were first introduced to the manipulator by a 10 minute long, written tutorial, culminating in a simple exercise. Each participant was tasked with creating five visualizations (Figure 2.5.1): two barcharts, one icicle layout, one treemap, and a custom tree layout. 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. For each task, candidates were given the same set of flexible blocks. Our study did not evaluate the selection of blocks. To complete each task, candidates had to produce the goal layout in 10 minutes or less using the manipulator.

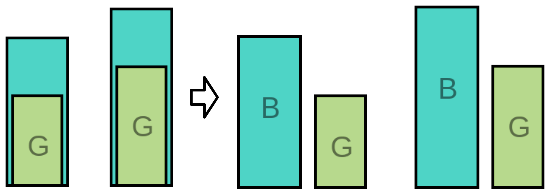
Results All participants but one solved each of the five tasks within the time limit. One participant was not able to complete the icicle graph. Results are summarized in Table 2.5.1. The treemap is a particularly interesting case: Participants found creative, unexpected ways to complete the task with 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

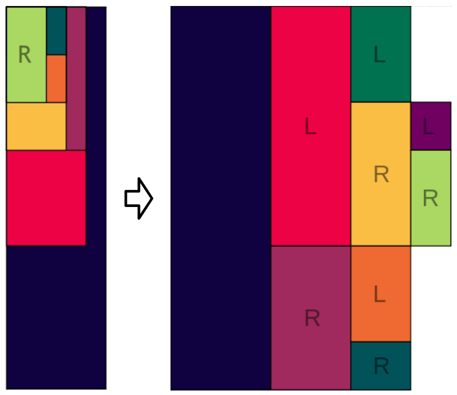
To make a fair comparison with manual constraints programming, we focus on the significant aspects of programming, such as resolving ambiguities and conflicts, while abstracting away irrelevant factors like language syntax. To do so, we built a second programming tool which mimics the relevant part of 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.



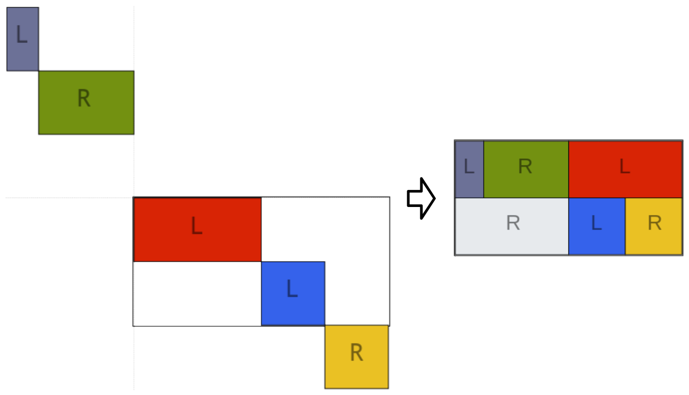
(a) Barchart A



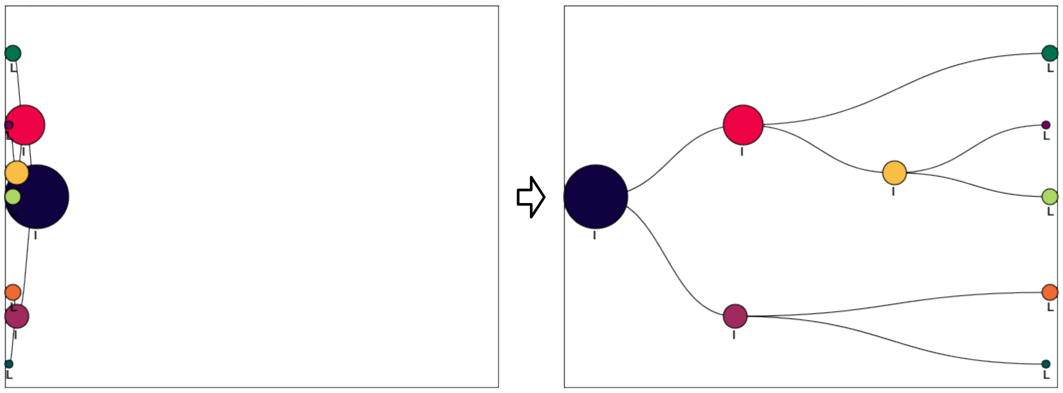
(b) Barchart B



(c) Icicles



(d) Treemap

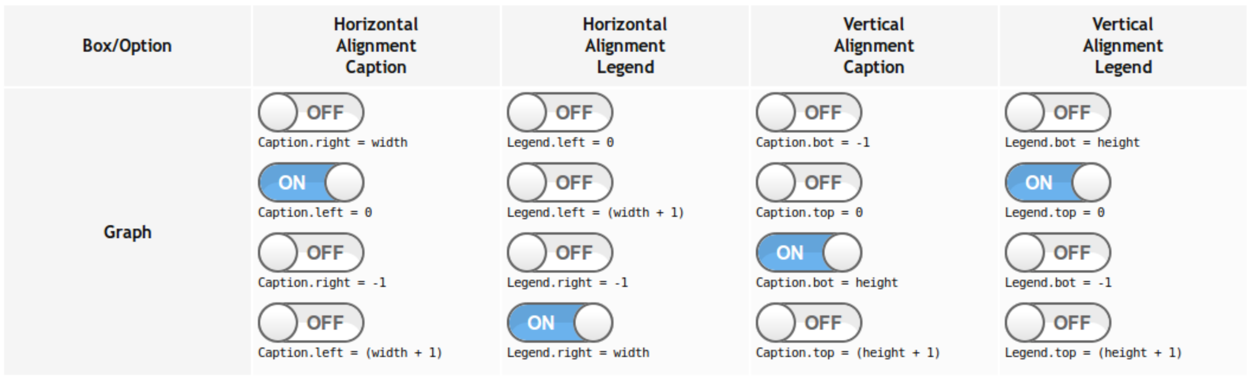


(e) Tree

Figure 2.5.1. The five tasks assigned to participants for both user studies. The left layout is the starting configuration, and the right layout is the goal configuration. The tasks were designed to be progressively increasing in difficulty.

	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 2.5.1. Non-programmer results. The columns indicate for each of the five tasks: 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.



Box/Option	Horizontal Alignment Caption	Horizontal Alignment Legend	Vertical Alignment Caption	Vertical Alignment Legend
Graph	OFF `Caption.right = width`	OFF `Legend.left = 0`	OFF `Caption.bot = -1`	OFF `Legend.bot = height`
	ON `Caption.left = 0`	OFF `Legend.left = (width + 1)`	OFF `Caption.top = 0`	ON `Legend.top = 0`
	OFF `Caption.right = -1`	OFF `Legend.right = -1`	ON `Caption.bot = height`	OFF `Legend.bot = -1`
	OFF `Caption.left = (width + 1)`	ON `Legend.right = width`	OFF `Caption.top = (height + 1)`	OFF `Legend.top = (height + 1)`

Figure 2.5.2. The table of switches of the button tool. Programmers enabled/disabled constraints by toggling them directly.

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, as shown in Figure 2.5.2. 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 the manipulator: the tool shows one possible layout augmented with axes of freedom representing the base of ambiguity for each partially constrained block.

Experimental Setup 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 the manipulator 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 the manipulator. 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.

Results 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. Results are summarized in Table 2.5.2. 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 difficulty, this was expected. The manipulator 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 the manipulator.

Discussion

Our first user study demonstrates that non-programmers can successfully design data visualizations using the manipulator, while the second study shows that programmers would also be more productive with PBM 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

	Button Tool				Manipulation Tool			
	Completion		**Time [s]**	**Steps**	**Completion**		**Time [s]**	**Steps**
Barchart A	16	(100%)	102 ± 50	11.9	16	(100%)	9 ± 5.6	3.0
Barchart B	16	(100%)	80 ± 59	12.7	16	(100%)	24 ± 15	5.8
Icicles	16	(100%)	362 ± 185	25.0	16	(100%)	37 ± 74	7.3
Treemap	15	(94%)	478 ± 278	25.9	16	(100%)	64 ± 38	10.9
Tree	15	(94%)	264 ± 160	30.3	16	(100%)	54 ± 52	6.9

Table 2.5.2. Programmers results. For each tool and for each of the five tasks, we report 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.

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 the manipulator. Non-programmers took on average 53% longer than the subset of programmers who started with the manipulator. 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 2.5.3 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 the manipulator.

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 to 16) shortly afterward. It took him eight attempts and a large amount of time—more than two thirds of the total time spent on this task—to resolve the conflict. Toward the end of the trace (step 23),

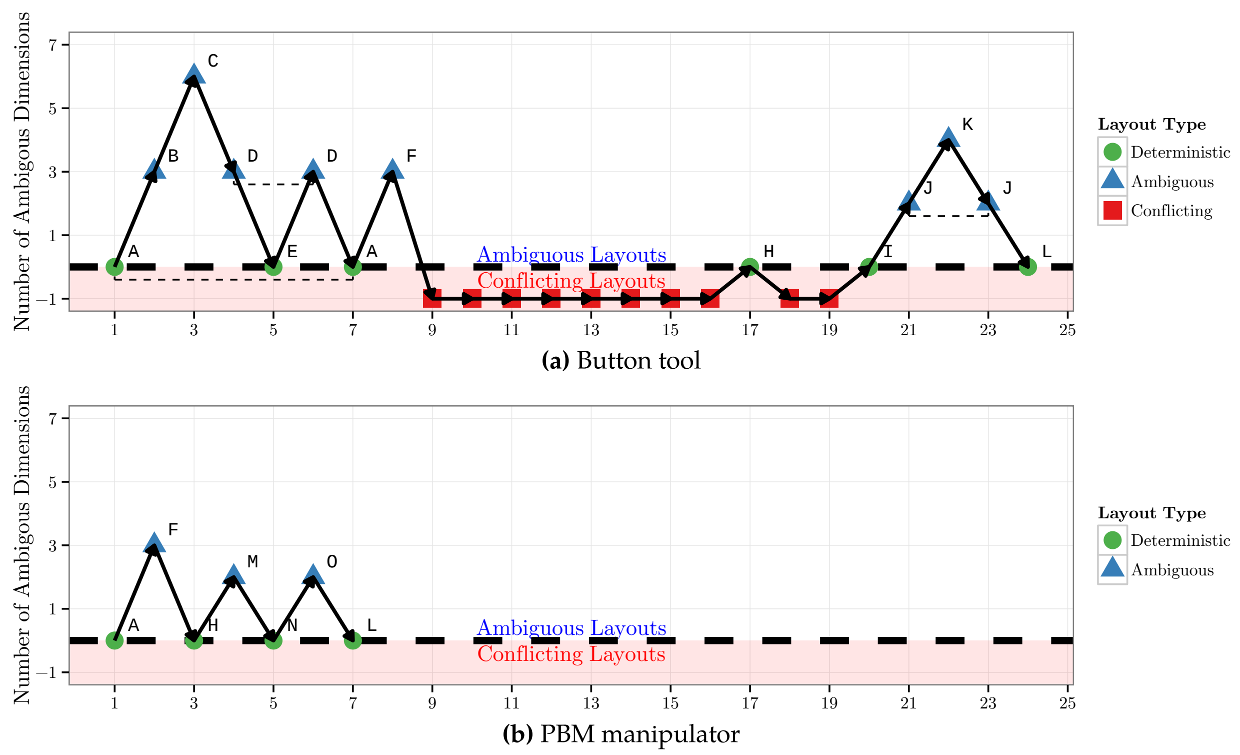


Figure 2.5.3. The paths through the configuration space on the icicles task from the same participant using both tools. Each configuration is uniquely identified by a capital letter. A thin dashed-line highlights configurations explored twice due to backtracking. Configurations along the x axis (0 ambiguous dimension) are deterministic. Conflicting configurations are represented with a negative number of ambiguous dimensions.

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 the manipulator. 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 the manipulator 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, PBM 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 the manipulator 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 the manipulator 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.

2.6 Related Work

This chapter builds on the foundations laid by PBD systems, GUI builders, and the recent work on fully automatic layout inference.

PBD Systems

Programming By Demonstration (Nevill-Manning, 1993; Cypher *et al.*, 1993) has been applied to a wide class of domains such as repetitive GUI interactions (Lau and Weld, 1999), text edits (Lau *et al.*, 2000), or string and integer programs (Singh and Gulwani, 2012b,a). Most these PBD systems rely on a refinement strategy in which the set of candidate programs is progressively narrowed down to a singleton as the user provides more demonstrations. Version space algebra (Lau *et al.*, 2003) provides the theoretical foundations for such refinement frameworks. As noted by Lau *et al.* (2003), refinement approaches are difficult to extend to domains in which demonstrations are noisy or imprecise, like layout. Due to the narrowing process, a single imprecision can rule out the desired program, causing the PBD process to fail eventually when it converges to an empty set of potential candidate.

In contrast, PBM was designed for noisy domains such as layout. Our approach does not follow a monotonic narrowing; it is exploration-centric. We focus on enabling users to navigate efficiently large spaces of candidates to find interesting layouts quickly. If a generalization or specialization is not fruitful, users can backtrack to a previously seen layout. As such, imprecisions can at worse cause detours but will never prevent users from reaching the goal layout.

However, we cannot offer termination guarantees. It is theoretically possible to loop over the same set of configurations forever.

GUI Builders

With GUI builders, users can construct user interfaces graphically by progressively adding widgets to a canvas. In particular, we note Peridot (Myers and Buxton, 1986) and its successor Lapidary (Myers *et al.*, 1989), Druid (Singh *et al.*, 1990), IBuild (Vlissides and Tang, 1991), Rockit (Karsenty *et al.*, 1993), and most recently ALE (Zeidler *et al.*, 2013). In such systems, each time a widget is added, new layout constraints fixing its position are inferred, sometimes with the help of semantic snapping (Hudson and Yeatts, 1991). More advanced systems such as ALE produce flexible GUIs which adaptively resize to occupy the space available. Naturally, GUI builders are tailored toward UI boxy or tabstop layouts (Hashimoto and Myers, 1992; Lutteroth *et al.*, 2008); it is unclear whether these techniques can be adapted to recursive layouts common in data visualizations, such as a radial tree.

PBM and GUI builders have orthogonal techniques to refine the layout specification. PBM starts from a full, complete but incorrect specification, and progressively adjust it by enabling/disabling constraints. In contrast, GUI builders start with an empty specification (an empty canvas) and progressively infer more constraints as widgets are added to the canvas, effectively completing the specification incrementally.

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 (Lutteroth *et al.*, 2008; Zeidler *et al.*, 2013) 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 (Weitzman and Wittenburg, 1994; Bateman *et al.*, 2001), or directly from user-drawn mock-ups (Sinha and Karim, 2013). 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.

Chapter 3

A Language of Constraints for Layout

In this chapter, we present L^3 (Language for Layout Languages), a constraint-based layout language. L^3 allows programmers to define new layout blocks. These blocks are flexible: non-technical users can configure their layout semantics by manipulation (Chapter 2). Once configured, L^3 layout specifications can be compiled into executable layout engines using grammar-modular synthesis (Chapter 4).

We start by describing three challenges in layout programming—expressibility, extensibility, and predictability—and justify our design decisions (Section 3.1). Then we give an overview of the key features of L^3 (Section 3.2) and showcase our language through examples (Section 3.3).

3.1 Motivation and Design Principles

Visual layout spans at least three domains, each with dedicated languages: CSS (Bos *et al.*, 2011; Lie and Bos, 1997) is an example of a document layout language; and QML (Digia/Qt Project, 2011) targets GUI layouts. Both are popular, mainstream languages in which designers declaratively define documents from existing blocks. These languages also provide layout engines capable of solving all such documents.

We describe L^3—a constraint-based, declarative language for visual layout. L^3 includes support for definition of new layout blocks and creation of small "domain-specific" languages of documents. The L^3 compiler can automatically generate layout engines tailored to such languages. L^3 can be used to create flexible blocks for Programming by Manipulation or to specify layout directly, without manipulation.

Expressiveness and Extensibility

Layout languages such as CSS and QML support the creation of documents, but they come with a fixed set of layout blocks, each defining intrinsic constraints on how an element is to be positioned relative to other elements. The inability to declare new blocks has at least three consequences:

- *Inextensibility* New layout blocks must be introduced by modifying the layout engine. QML permits C++ procedures that compute custom sizing or positioning. In a web browser, the layout engine cannot be extended, so new layouts are created with custom JavaScript code, bypassing the native layout engine. Neither approach lends itself to easy prototyping of new layouts.

- *Rigidity* Programmers often resort to "side-effect" layout programming. Rather than directly expressing properties of the desired layout, such as relative positions of the visible elements, programmers must often express constraints *indirectly*, by composing the document from available blocks. The intent is that satisfaction of the indirect properties entails the desired properties. Unfortunately, the entailment is hard to reason about, in part because the constraints are implicit in the definition of the blocks.

  Furthermore, the entailment may not hold in general, leading to unpleasant surprises (Figure 3.1.1). In CSS, a classic illustration of this phenomenon is grid layouts implemented with floats. The goal is to place ten icons in a 5x2 grid. We set icon sizes and column widths to fit five icons on one line, with the remainder overflowing to the next line. This is an indirect way to express our goal. These constraints do create a grid design—until an icon must be resized to accommodate an oversized caption, which pushes icons to the third line, ruining our intended grid design.

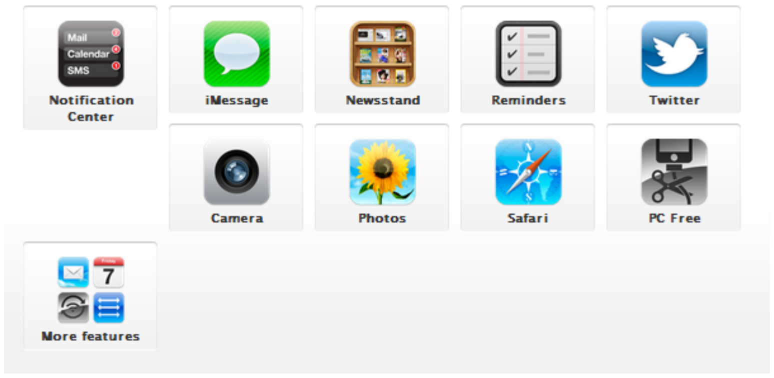


Figure 3.1.1. Surprises arise when indirect layout constraints do not entail the desired specifications. The oversized icon shifted the second row, rather than localizing the effect of its oversize.

- Limited Expressiveness The fixed layout computation strategy places restrictions on layout designs. While new layout behaviors can be created through the composition of the available blocks, expressiveness by composition is limited. In particular, layout engines that achieve efficiency by fixing a static traversal schedule of the document—for example, CSS engines compute width before height—may rule out some desired layouts.

As an example of a CSS-inexpressible layout, assume you want to lay out a document with a sidebar which must be computed to be wide enough to display all of its content on a single screen without overflowing the bottom of the screen (Figure 3.1.2). The contents of the main panel, however, are allowed to overflow. To compute such a layout, one would first compute the width of the sidebar, given the screen height, and then compute the main area height, given the sidebar width. Surprisingly, such a simple design is impossible to implement with CSS, which must always compute heights as functions of widths.

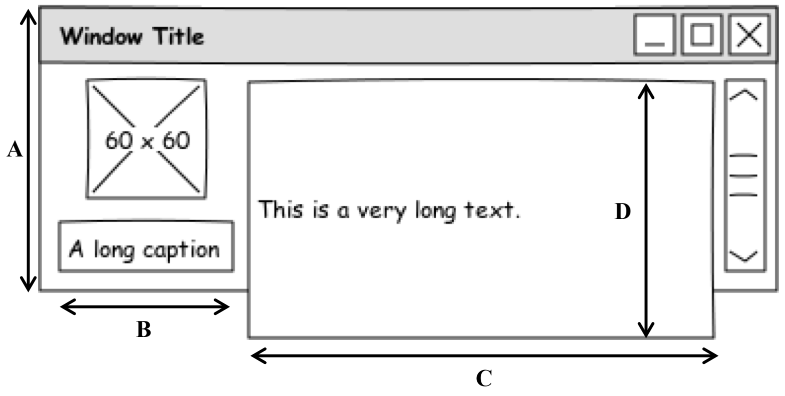


Figure 3.1.2. A Layout inexpressible in CSS. The width of the sidebar (B) is computed from the height of the screen (A) so that its content fits in a single screen. Then renaming width (C) allocated for the main panel, whose height (D) may overflow if necessary.

Our language, L^3, allows definition of new layout blocks, not just documents. Furthermore, by specifying which nestings of blocks are legal, programmers can create small languages of documents, specialized for one particular type of layout. Layout needs vary across domains. A magazine like NewYorker.com emphasizes perfect text layout, while a web-mail like Gmail.com is tabular and uses sophisticated scroll-boxes. We believe that small, domain-specific, layout languages can avoid the limitations of large—one size fits all—languages by providing very specific building blocks.

L^3 offers the following advantages over mainstream layout languages:

- *Explicit Constraints* New blocks are programmed by setting constraints on arbitrary properties of visual elements. Desired layouts can thus be expressed directly, avoiding the invisible cascade of consequences arising from built-in implicit constraints.
- *Non-directional constraints* When a constraint states that "A is aligned with B," it does not specify whether the position of A is computed from B or vice versa. This (global) determination is delegated to the L^3 compiler. Our constraints are (non-directional) relations, rather than (directional) functions, thus freeing programmers from reasoning about artifacts of computation, raising the level of abstraction. For the sidebar problem, for instance, it is sufficient to state that the height of the sidebar is no greater than the

screen height. The L^3 compiler will produce a layout engine that will, from the screen size, compute the sidebar width, from which the text height will be computed.

Non-directionality is also useful in interactive layouts, where mutual constraints are common and the computation may flow in either direction, depending on the user's interaction. For instance, in a scroll-box, moving the content moves the scrollbar, while moving the scrollbar moves the content (Figure 1.4.2). This example is detailed in Section 1.4.

- *Layout Engine Generation* New layout blocks are introduced declaratively, without extending an operational layout engine. The L^3 compiler generates the corresponding layout engine automatically.

Predictable and Diagnosable Constraints

Existing layout languages use constraints that can be dropped when they cannot all be satisfied (see Section 2.1). General-purpose languages like CSS are large with complex and sometime obscure interactions between constraints embedded in blocks. When undesirable interactions result in inconsistencies, the engine silently drops constraints. This behavior renders programming with CSS unpredictable and hinders debugging.

Document layout has been specified with constraints in prior research languages (see Section 3.4). These languages cast layout as an optimization problem, either by maximizing a utility metric or by trying to satisfy as many constraints as possible. Optimization- based approaches are more expressive—they can capture spring networks, unlike satisfiability constraints. While these semantics make constraint relaxation systematic, the designer still cannot entirely predict the resulting layout due to spring-effects (see Section 2.1 and Figure 2.1.1).

We propose to specify layout with *satisfaction constraints*, which means that all constraints must be satisfied. (L^3 programmers are still allowed to specify alternative constraints with disjunctions.) Casting layout as a constraint satisfaction problem offers the following two benefits:

- *Predictability* Satisfiability constraints predictably control the resulting layout. Because each constraint is always satisfied, the programmer is assured that "what you state is what you get".

- *Analysability* Satisfiability facilitates static analysis of L^3 constraints, enabling the manipulator to prevent conflicts, summarize ambiguities, and efficiently compute both generalizations and specializations. When programming without the manipulator, a challenge is to ensure that the constraints uniquely determine the layout. We detect at compile time when they do not, and given a document, we present two distinct layouts to the programmer. Visualizing these two layouts helps with understanding which constraints are missing (Figure 3.3.1).

3.2 Overview of L^3

In this section, we present a high-level overview of L^3 from a programmer's perspective. Recall that documents are trees of nodes; each node is labeled with a block. Blocks define the layout semantics of document nodes: typically how each node's positions and sizes are computed. As such, the block of a node is akin to its type. Each block gives a visual appearance and attributes to the document node. Some attributes can be marked as *input*; these are run-time constants unknown at compile time, *e.g.* the size of an image or the size of the top-level window. The layout semantics of blocks are defined by placing constraints on attributes. Solving a document amounts to computing the value of all attributes, given input values, in accordance with the blocks' semantics.

A layout specification consists of two parts: (i) a definition of the semantics of each layout block; and (ii) a description of which nestings of nodes are allowed in documents. Together, both parts constitute a definition of a language of documents and their layout semantics.

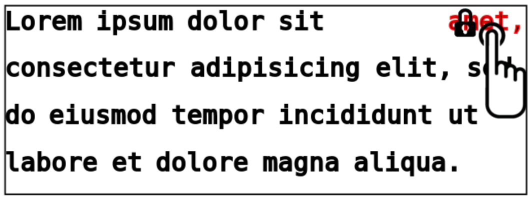
Block Semantics

Each block defines a set of attributes and places constraints over them. L^3 constraints are *local*—only variables from the direct parent or children in the document hierarchy can be referred to.

L^3 constraints are also *non-directional*—they leave the flow of computation unspecified, up to the compiler. By capturing multiple flows of computation at once, non-directionality enables very concise descriptions of blocks with mutually dependent attributes. For instance, we can fix the aspect ratio of an image while leaving open whether the height is computed from the width or vice-versa.

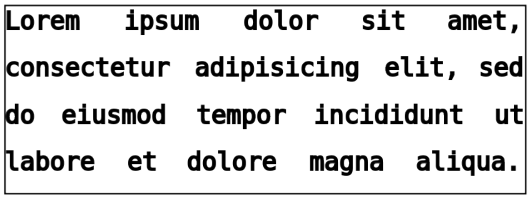
Under the hood, our language of constraints is based on SMT theories (Barrett *et al.*, 2010) which provide an expressive set of primitive constructs. The techniques presented in this thesis are independent of the logical theories used to express constraints. Both the manipulator (Section 2.4) and the L^3 compiler (Chapter 4) only require a decision procedure capable of answering satisfiability queries over constraints. Our examples and implementation rely on polynomial equations and linear inequalities over reals augmented with basic trigonometric functions as well as operators max and min. Empirically, we found such constraints rich enough for a wide class of layouts and visualizations. For instance, polynomials are frequently used to capture ratios of relative sizes as in the scroll-box (Figure 1.4.2). Figure 3.2.1 illustrates the power and versatility of our constraints with an example of flow-layout with justification.

Blocks are either *configured* (all constraints are mandatory), or *flexible* (there are optional/alternative constraints). By bundling optional constraints, flexible blocks make their specification customizable by non-technical users using Programming by Manipulation. The manipulator compiles flexible blocks into configured blocks where all optional constraints either have been



Lorem ipsum dolor sit amet,
consectetur adipisicing elit, s
do eiusmod tempor incididunt ut
labore et dolore magna aliqua.

(a) Left Justification



Lorem ipsum dolor sit amet,
consectetur adipisicing elit, sed
do eiusmod tempor incididunt ut
labore et dolore magna aliqua.

(b) Full Justification

Figure 3.2.1. A complex flow layout. We illustrate the expressibility of our constraints by capturing text-layout. Proper typesetting of text block is arguably the most complex part of layout. The difficulty resides in finding the right place to insert line-breaks and adjusting the word spacing to obtain an aesthetically pleasing result (Achugbue, 1981). Both line-break insertions and word-spacing are computed as part of the layout. In the figure on the left, the user is making a WiW manipulation to go from left justification to full justification.

selected and made mandatory or have been discarded. Configured blocks can also be declared directly by programmers to specify layout without PBM.

Traits 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, L^3 supports composable modules of constraints called *traits*. There are two kinds of traits: *mandatory* traits bundle a set of constraints conjunctively, whereas *optional* traits introduce flexibility by bundling constraints disjunctively: any subset of their constraints (including the empty set) may be satisfied.

We show below one of the most frequently used mandatory traits which defines the CSS box model. Since our constraints are non-directional, equal symbols represent equalities, not assignment.

```
mandatory trait CssBoxModel {
    top + total_height = bot
    left + total_width = right
    x = parent.content_x + left
    y = parent.content_y + top
    margin_left + border_left + padding_left = content_x - x
    margin_top + border_top + padding_top    = content_y - y
    padding_left + width + padding_right = inner_width
    padding_top + height + padding_bot    = inner_height
    border_left + inner_width + border_right = border_width
    border_top + inner_height + border_bot    = border_height
    margin_left + border_width + margin_right = total_width
    margin_top + border_height + margin_bot    = total_height
}
```

Listing 3.2.1. An implementation of the CSS box model in L^3.

As example of an optional trait, we show a vertical grouping trait proposing two ways of computing the height of container so that it encompasses both of its children.

```
1  optional trait VerticalGroup {
2      height = max(child0.total_height, child1.total_height)
3      height = child0.total_height + child1.total_height
4  }
```

Listing 3.2.2. An optional trait containing two alternative constraints.

For instance, in Figure 2.1.4a, the column blocks grouping one green bar and one blue one vertically use the first constraint (Line 2) initially. This constraint is then disabled by a WiW manipulation (Figure 2.1.4b) and finally, the second constraint (Line 3) is chosen by the user (Figure 2.1.4c).

Trait Composition The result of composing traits together is a new trait with the union of each trait's attributes and the conjunction of each trait's constraints. Trait and block are the same language construct; the distinction is only there to let programmers state whether a bundle of constraints is complete or only partial and meant to be composed.

Mandatory traits can also be composed optionally, creating a bundle of constraints which can be enabled or disabled at once by manipulation. This feature is useful when a group of related and interdependent constraints must be treated as an atomic unit. In the example below, *myBlock* is constructed by compositing three mandatory traits. *CssBoxModel* is composed conjunctively, but *VerticalStack* and *TopAlign* are composed optionally, as denoted by the trailing *?* in the block definition (Line 11).

```
 1  mandatory trait VerticalStack {
 2    child0.top = 0
 3    child1.top = child0.bot
 4  }

 6  mandatory trait TopAlign {
 7    child0.top = 0
 8    child1.top = 0
 9  }

11  block myBlock with CssBoxModel, TopAlign?, VerticalStack?, ...
```

Listing 3.2.3. Conjunctive and disjunctive compositions of traits.

In fact, optional traits are syntactic sugar for multiple optional compositions of mandatory traits containing a single constraint.

Language of Documents

By stating which nestings of blocks are legal, we define a language of documents. Such languages are specified with a regular tree grammar[1]. Each terminal of the tree grammar corresponds to a block-labeled node. Consequently, every derivable tree forms a document. Listing 1.4.2 shows the definition of a treemap language directly with a tree grammar.

Fundamentally, the languages of documents can be viewed as relational attribute grammars. A relational attribute grammar is an attribute grammar with constraints (*i.e.*, relations) instead of update functions (Knuth, 1968; Deransart and Maluszynski, 1985).

L^3 also has a *widget* construct which can be used to define languages from recursive document trees. Widgets provide an alternative way to specify a language of documents which does not requires knowledge of the grammar formalism. An equivalent definition of our language of treemaps using widgets is shown below.

```
 1  block root with ...
 2  block hdiv with ...
 3  block vdiv with ...
 4  block leaf with ...

 6  widget Hdiv() {
 7      layouts {
 8          leaf
 9          hdiv { Vdiv() Vdiv() }
10      }
11  }

13  widget Vdiv() {
14      layouts {
15          leaf
16          vdiv { Hdiv() Hdiv() }
17      }
18  }

20  language treemaps {
21      root { Hdiv() }
22  }
```

The *layouts* section of each widget declares subtrees of nodes which will be instantiated in place of the widget to create a document. As such, each such subtree is akin to the production rule of a grammar. For example, the *Hdiv* widget represents either a *leaf* (Line 8) or an *hdiv* node whose children are a pair of *Vdiv* widgets (Line 9). In essence, widgets make trees of nodes a first class construct. Widgets can be recursively referenced, enabling the creation of languages

[1]Regular tree-languages can be viewed as the set of derivation trees of a context-free word grammar (Comon *et al.*, 2007).

containing an unbounded number of documents. In the example shown above, *Hdiv* and *Vdiv* are mutually recursive.

3.3 L^3 by Example

We showcase the features of L^3 in three examples drawn from data visualizations and web page layout. In particular, we reflect on our design principles and evaluate L^3 empirically along the following two axes:

- *Expressiveness* Can L^3 express layout not realizable in mainstream languages such as CSS? Does non-directionality keep layout specifications concise and close to what a natural language specification might be?

- *Programmability* Do traits provide a meaningful decomposition of layout semantics leading to code reuse? Does our satisfiability approach to layout facilitate the identification of errors?

We evaluate our compiler and the performance of the layout engines generated in Chapter 4.

Treemap

Treemaps are a popular visualization in financial circles to compare the market values of companies (see Section 1.2). Figure 1.2.1 shows a (small) treemap document and its layout. Recall that treemaps are constructed from three main blocks: *hdiv*, *vdiv*, and *tile*.

Let us first write down the layout specifications in English: (i) *hdiv* and *vdiv* partition their space horizontally/vertically among their children; (ii) the area of a *tile* is proportional to its capitalization; and (iii) the capitalization of an internal node is regarded as being the sum of the capitalizations of its children.

We meet the first requirement by composing two traits: *LeftAligned* and *VerticalStack* for the vertical divider (block *vdiv*); *TopAligned* and *HorizontalStack* for the horizontal one. The second and third requirements encode the essence of treemaps. We express them by introducing one new trait: *TreeMap*.

```
mandatory trait LeftAligned {
    child0.left = 0
    child1.left = 0
}

mandatory trait TreeMap {
    scale * cap = height * width
    cap = child0.cap + child1.cap
}
```

```
11  configured block vdiv with LeftAligned, VerticalStack,
12                              SumHeights, EqualWidths,
13                              TreeMap, SimpleBoxModel,
14                              RelCartesian
15  configured block hdiv with ...
16  configured block tile(@cap) with TreeMap, SimpleBoxModel,
17                                    RelCartesian
18  configured block root with CartesianRoot {
19      3 * height = 4 * width
20  }
```

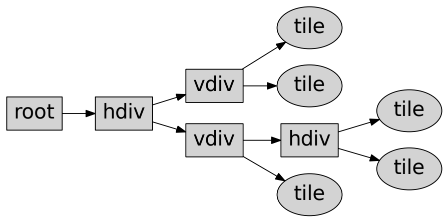
Listing 3.3.1. The specification of a treemap in L^3. The "@" symbol in the declaration of the *cap* attribute in block *tile* denotes a runtime input. In the *TreeMap* trait, *scale* is a constant converting dollars into squared pixel. *VerticalStack* is defined in Listing 3.2.3.

We also compose *RelCartesian* to set up relative coordinates (Listing 1.4.1) and *SimpleBoxModel*, a simpler version of *CssBoxModel* (Listing 1.4.1)

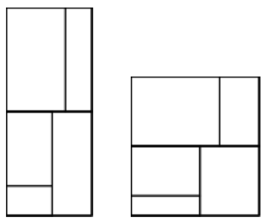
Discussion Treemaps are particularly hard to build with document layout languages like CSS. Computing sizes and positions of each node based on their area is incompatible with assumptions hardwired into these languages. We are not aware of any treemap implementation on the web which does not use a handcrafted JavaScript layout engine.

The translation from informal English specification to constraints was straightforward. Using traits, our code closely follows the structure of the informal specification by using the same decomposition. We hope that our language allows specifications that are closer to the human thought process, and that consequently our language is more approachable and easier to use than other layout languages.

In our first attempt to express a treemap, we forgot to specify the desired aspect ratio in the *root* block (Line 19). When this constraint is omitted, documents are ambiguous: they have many valid layouts, one per aspect ratio. To help programmers debug layout specifications, the L^3 compiler can exhibit some of the possible layouts as visual guidance. Figure 3.3.1 shows the document as well as two possible layouts produced by our compiler. Upon seeing the two layouts, it immediately became clear what the source of the ambiguities was, and fixing the bug was easy. We believe that providing designer-friendly debugging information is key to making constraint-based languages accessible to a wider audience of less experienced programmers. By rejecting our first incorrect attempt at the specification of treemaps, L^3 prevented the common scenario of ambiguous documents showing discrepancies when laid out on multiple platforms. Such differences in layout often appear inexplicable to designers. The ability to detect these bugs statically is one of the strengths of our satisfiability approach to layout.



(a) One document



(b) Two layouts

Figure 3.3.1. Debugging under-constrained languages. The two treemaps on the right have the same area but a different aspect ratio.

Flexible Treemap

In the previous section, we constructed a treemap from configured blocks, without using manipulation. Let us now create a treemap by defining flexible blocks whose specifications can be customized with PBM.

We replace mandatory traits with optional traits. For instance, we replace *LeftAligned* with two traits bundling constraints for horizontal alignment, one for each child: *HAlignChild0* and *HAlignChild1*.

```
optional trait HAlignChild0 {...}
optional trait HAlignChild1 {
    child1.left = 0
    child1.right = width
    child0.right = child1.left
    child0.left = child1.right
    child1.right = 0
    child1.left = width
    child1.left = child0.left
    child1.right = child0.right
}

optional SimpleHeights {
    height = child0.total_height = child1.total_height
    height = max(child0.total_height, child1.total_height)
    height = child0.total_height + child1.total_height
}

block hdiv with HAlignChild0, HAlignChild1,
                VAlighChild0, VAlignChild1,
                SimpleHeights, SimpleWidths,
                TreeMap, SimpleBoxModel, RelCartesian
block vdiv with HAlignChild0, HAlignChild1,
                VAlighChild0, VAlignChild1,
                SimpleHeights, SimpleWidths,
                      TreeMap, SimpleBoxModel, RelCartesian
```

Listing 3.3.2. The definitions of flexible treemap blocks. The specifications of *hdiv* and *vdiv* are customized by manipulation.

Notice that the *hdiv* and *vdiv* blocks have identical definitions. They share the same universe of optional constraints. The semantic differences between these two blocks are established by manipulation. Using the manipulator, the user will configure these blocks with distinct sets of constraints.

The only trait specific to treemap visualizations is the *TreeMap* trait. All other traits embody concepts reused across many other layouts and visualizations.

Sun Burst

Our second visualization uses a polar coordinate system to lay out concentric disks (Figure 3.3.2). For instance, such "sun burst" layouts are used to represent file system usage. In this case, the document is the filesystem tree, inner nodes are directories and leaves are files. Each file is annotated with its size. We show below the (simplified) definition of the *directory* block.

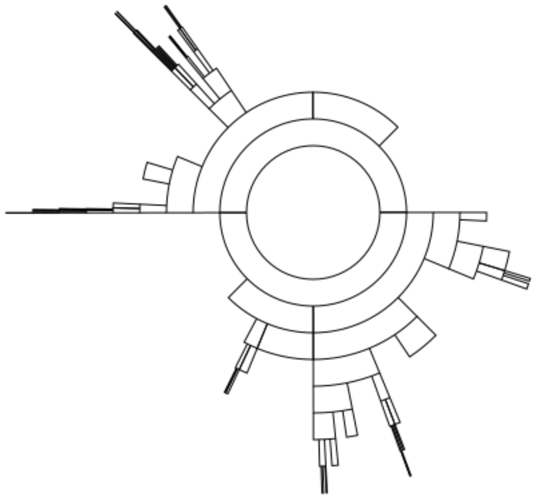


Figure 3.3.2. A visualization of file system usage using a polar layout.

```
mandatory trait Polar2Cartesian {
    x = radius * cos(angle)
    y = radius * sin(angle)
}

mandatory trait Wedge {
    children.radius = radius + height
    span * parent.size = parent.span * size
    child0.angle = angle
    child1.angle = angle + child0.span
}

configured block dir with Wedge, Polar2Cartesian {
    size = child0.size + child1.size
    height = 20
}
configured block file(@size) {...}
```

Listing 3.3.3. A sun burst visualization in L^3. The size of each file is a runtime input. The height of each wedge could also have been a runtime input.

With this second sun burst example, we show that L^3 makes no assumption about the layout model. We are not restricted to "boxy" layouts, unlike GUI languages like QML or document languages like CSS. L^3 constraints enable concise descriptions of new layout elements.

CSS3 FlexBoxes

For our last example, we look at a feature of graphical user interfaces: grid-based layouts. The draft of the new revision of CSS includes a proposal of a new box model targeted specifically at GUIs: *flexboxes* (Deakin *et al.*, 2011). One of the goals of the proposal is to introduce a grid-based layout system with well-defined semantics. Each flexbox divides its space into either columns or rows. Each column (or row) specifies its desired width (or height) with a *flex* attribute. The *flex* attribute is a runtime input and controls the allocation of the widths of columns following a weighted sum distribution. Flexboxes can be implemented easily with L^3. We show below the key constraints encoding vertical flexibility.

```
trait VerticalFlex {
    total_flex = child0.flex + child1.flex
    total_flex * child0.height = child0.flex * height
    total_flex * child1.height = child1.flex * height
}

block vflex(@flex) with CssBoxModel, VerticalFlex, ...
```

3.4 Related Work

L^3 builds upon a long history of declarative layout languages. Constraints have been used in many languages to specify layout. See Hurst *et al.* (2009) for a recent overview of the field. In particular, we note the constraint-based formulations of two existing and widely used layout languages: CSS and SVG (Badros *et al.*, 1999, 2001b).

In the GUI domain, Freeman-Benson (1993) explains how to convert an existing user interface to constraints. Numerous constraint-based specifications for GUI have also been proposed (Myers, 1988; Feiner, 1988; Myers *et al.*, 1989, 1990; Helm *et al.*, 1992; Maloney, 1992; Lutteroth *et al.*, 2008). In document layout, constraints have also been extensively been studied, most notably for web pages (Borning *et al.*, 1997, 2000; Badros *et al.*, 1999; Hurst *et al.*, 2003). In particular, Meyerovich (2013) formalizes the core of CSS with directional constraints. Another line of work investigates adaptive document layout, capable of adjusting to media of various sizes (Jacobs *et al.*, 2003; Schrier *et al.*, 2008). For data visualizations, Protovis (Heer and Bostock, 2010) and its successor D3 (Bostock *et al.*, 2011) are two popular languages. The problem of specifying three dimensional layouts has also been explored (Elliott *et al.*, 1994).

Finally, layout can also be specified declaratively from topological descriptions (Weitzman and Wittenburg, 1993, 1994; Di Iorio *et al.*, 2008).

Much of the aforementioned work focuses 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 either by maximizing a utility metric or by satisfying as many constraints as possible.

We chose to cast layout as a satisfaction problem with the following benefits: satisfiability is a simpler problem in terms of computability; rich constraints such as polynomials, which are common in visualizations, become tractable. Satisfaction also enables a deeper level of analysis: the manipulator relies on satisfiability queries to prevent conflicts, summarize ambiguities, and efficiently compute both generalizations and specializations. Furthermore, satisfiability constraints let us leverage existing program synthesis techniques to generate layout engines automatically.

Chapter 4

Grammar-Modular Synthesis

This chapter examines how constraint-based layout languages, such as L^3 (Chapter 3), can be solved efficiently. The technique proposed here is based on program synthesis—the problem of translating high-level specifications into programs directly executable. We present Grammar-Modular (GM) synthesis, an algorithm for synthesis of programs from tree-structured relational specifications. We show how GM synthesis can compile L^3 to efficient tree-traversal layout engines.

We start by introducing the program synthesis problem and then discuss the challenges posed by our domain, layout (Section 4.1). In Section 4.3, we introduce GM synthesis for single relations. Then we generalize our algorithms to grammars of relations and discuss the completeness of our approach (Section 4.4). Finally, we present our experimental results on synthesis of layout engines (Section 4.5).

Even though the challenges posed by layout led us to develop GM synthesis, the techniques presented in this chapter are generic. We use GM synthesis to compile L^3 layout specifications to tailored layout engines but the algorithm is applicable to any hierarchical specification expressible as a relational attribute grammar; it could, perhaps, be used to raise the level of abstraction in compiler construction, which is often specified as a functional attribute grammar.

4.1 From Program Synthesis to GM Synthesis

By raising the level of abstraction, automatic synthesis of programs from specifications has the potential to make programming easier. Program specifications can often be stated as a relation between inputs and outputs, for instance, with pre and post conditions. Then we can synthesize a program by turning the relation (specification) into a function (program) according to an input/output (known/unknown) partition of the relation's variables. This type of program synthesis is usually called functional synthesis. In this chapter, we focus on relations expressible in propositional SMT logics (Barrett *et al.*, 2010).

A key benefit of functional synthesis is enabling programmers to use declarative constraints (*i.e.*, relations) without incurring the cost of solving constraints. Generally, constraints are computed at runtime by a solver for a particular input (*i.e.*, initial conditions). By synthesizing functions from constraints, we obviate the need for constraint solving at runtime and shift this cost to compilation time. Intuitively, the synthesized functions execute only value propagations and bypass the backtracking search performed by constraint solvers. In situations where the same constraint system must be solved multiple times with varying inputs, the same synthesized functions can be reused, making the performance gains brought by synthesis even more attractive.

Our goal is to scale functional synthesis to large relations. At heart, functional synthesis is a quantifier elimination problem: we eliminate variables from the relation until all outputs can be expressed only in terms of inputs. Recent work such as Comfusy (Kuncak *et al.*, 2010) has brought program synthesis to mainstream compilers. Comfusy is an extension of Scala allowing relational constraints in functional programs. In essence, Comfusy translates the execution of a quantifier-elimination procedure on a particular relation into SMT formula whose models capture the key steps of quantifier elimination. Ultimately, given such steps, Comfusy can construct functions computing the outputs. In fact, such functions can be viewed as specialized solvers tailored for one particular constraint system. In practice, the efficiency of the translation and scalability of SMT solvers limit the size of relations which can be functionalized, *i.e.*, for which we can compute (executable) functions. Empirically, we found that quantifier-elimination based approaches do not scale to the large specifications of our domain, document and visualization layout.

Synthesis of Layout Engines

Layout specifications are naturally expressed with constraints (Hurst *et al.*, 2009). Constraint-based languages such as L^3 (Chapter 3), CCSS (Badros *et al.*, 1999), and ALE (Zeidler *et al.*, 2013) are both powerful and versatile. Even CSS, the ubiquitous web template language, relies on constraints, although in a more restricted and indirect manner (Hurst *et al.*, 2009). As shown in Chapter 2, constraints also enable the inference of layout specifications directly from user demonstrations (Myers *et al.*, 1989; Hottelier *et al.*, 2014).

Layout engines compute attributes such as the sizes and positions of all visual elements from input attributes, which are runtime constants (*e.g.*, the window size). When layout is specified with constraints, solving them quickly enough (less than half a second) to enable smooth user interactions is a major technical challenge. Today, the average webpage has over a thousand elements, each with dozens of attributes (Souders, 2013). Since layout engines are executed numerous times, for instance, to handle resize-window events or to adapt to new data values, the potential cumulative runtime savings from synthesizing specialized "function" solvers are large. For these reasons, we believe that automatic generation of layout engines is a prime target for functional synthesis.

Layout can be computed either by solving constraints at runtime (*i.e.*, after input values are provided) with a general-purpose solver, or by handcrafting a solver (engine) tailored to a particular type of visualization. Handcrafted solvers are usually implemented as a bounded set of tree traversals over a hierarchical document labeled with constraints. Today, general-purpose solvers are too slow for interactive settings (up to 200x slower than handcrafted solvers, see Section 4.5). As such, all browsers and most visualization libraries such as D3 and Protovis (Heer and Bostock, 2010; Bostock *et al.*, 2011), rely on handcrafted solvers. However, writing such solvers is time-consuming. As a result, trying out design ideas is expensive.

Functional synthesis promises to combine the performance of handcrafted engines with the ease of use of constraint solvers. By applying functional synthesis to the relational specification of layouts, we generate a solver specialized for a particular set of constraints. In essence, we automate the tedious optimizations currently performed manually by visualization programmers. Ultimately, layout is a domain in which functional synthesis could have a significant impact.

However, scaling synthesis to large relations is a challenge; so far, synthesis has mostly been limited to producing program fragments. Our experiments show that Comfusy scales to 100 variables at most. However, the relations describing layout can range over 10^4 program variables, more than one order of magnitude larger than what state-of-the-art synthesis tools can handle. We present Grammar-Modular (GM) synthesis, a technique to scale functional synthesis to large and hierarchical relations, such as data-visualization specifications.

Modular Synthesis

To scale synthesis to large relations, we rely on the presence of a hierarchical structure to trade completeness for scalability. Specifications can often be written as conjunctions of smaller relations. We exploit this structure to decompose the synthesis problem into smaller subproblems whose solutions can eventually be combined together to functionalize the overall relation. We call this technique *modular* synthesis.

Layout specifications, for instance, naturally give rise to a hierarchical decomposition. The data to be laid out is commonly represented as a tree of nodes—the document. Each document node is encoded as its own conjunct. We can apply synthesis on each node individually and then combine the results together to create an engine computing the layout for the whole document.

The key technical challenge of modular synthesis is the construction of a global function satisfying the overall relation, from the local functions produced by applying functional synthesis on each sub-relation. We cast the creation of a global function as (function) compositions of local functions. By doing so, we trade completeness for efficiency: modular synthesis cannot perform deduction across decomposition boundaries (only function composition), so a relation that can be functionalized globally may not be functionalizable in a modular way. The smaller relations may not be functional and hence the necessary local functions cannot be produced.

Grammar-Modular Synthesis

So far we have outlined how modular synthesis can generate functions solving one particular relation. In the layout domain, each type of document node instantiates the same local constraints. For performance, we would like to avoid applying the GM synthesizer anew for each relation (*i.e.* document). We would like to avoid not only re-synthesis of local functions, but also the expensive composition of the global function, and simply thread local functions together based on the tree structure of the document.

Imagine you have written a specification of a simple visualization: a barchart. The synthesis techniques presented so far would generate an engine specific to this particular barchart. That is, the engine would only function on a single document. Our constraints bind a fixed set of variables; relations for barcharts with a different number of bars have a different number of variables. If the dataset changes to require more bars, a new engine must be synthesized.

To be practically useful, we must synthesize engines capable of adapting to such changes by handling multiple documents, each with a particular number of bars, for instance. That is, the engine must be generic enough to solve multiple relations, even an unbounded number of relations. Figure 2.1.3 shows two documents from a language of treemaps (defined in Listing 1.4.2) laid out by the same synthesized engine.

In essence, we generalize program synthesis to accept not a fixed relation but a language of relational specifications. We restrict ourselves to *regular* tree-languages of relations whose variable sharing structure forms a tree. By doing so, we can represent the synthesized program as a set of functions whose composition is syntax-directed by the structure of the relation. Fundamentally, we are converting a relational attribute grammar[1] into a traditional, functional attribute grammar that is statically schedulable. We call this technique *grammar-modular* (GM) synthesis. Given a language of relations and an input/output partition of its variables, we synthesize a functional attribute grammar capable of computing the outputs of any relation in the language. More technically, we generalize modular synthesis to grammars of relations by handling alternative and recursive productions. To guarantee that our functional attribute grammars are statically schedulable, we reject grammars with cyclic dependencies between attributes, thereby forbidding fixed-point computations.

Regular tree-languages include most layout languages, including data visualizations. GM synthesis enables automatic generation of layout engines from specifications of layout languages. Such languages define both syntactically legal documents and their layout semantics. Each such grammar defines a language of documents and its layout semantics. The layout engine, a functional attribute grammar, computes all document attributes (*e.g.*, sizes, position) given runtime-inputs (*e.g.*, window size), which are given as values of some attributes. Eventually, the layout engine can be scheduled to tree traversals with the same form as handcrafted engines. In fact, from the same relational specification, we can synthesize distinct layout engines, depending

[1]A relational attribute grammar is an attribute grammar with constraints (*i.e.*, relations) instead of update functions (Knuth, 1968; Deransart and Maluszynski, 1985).

on which attributes are inputs and which attributes are computed, which is useful in interactive situations. For instance, in a scroll-box, when the user moves the slider, the document position can be computed from the slider and vice versa. Each such user interaction triggers a different flow of attribute updates, but maintains the same constraints.

4.2 Compiler Architecture

Recall that L^3 layout specifications consist of two parts: (i) a definition of the layout semantics of each block; and (ii) a description of which nestings of nodes are allowed in documents. Together, both parts constitute a relational attribute grammar (Knuth, 1968) which defines a language of documents together with layout semantics (Section 3.2).

Given a set of blocks and a tree grammar, our synthesizer outputs a layout engine, in the form of a functional attribute grammar (Knuth, 1968), capable of computing the layout of all derivable documents. Figure 4.2.1 shows the architecture our L^3 compiler.

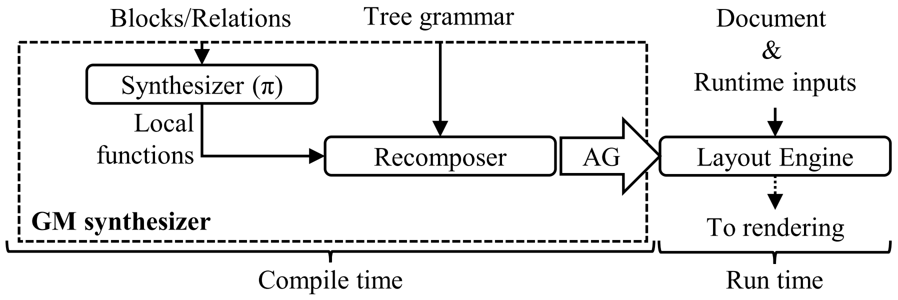


Figure 4.2.1. The architecture of our L^3 compiler. The first step of GM synthesis—decomposition—is not shown. The attribute grammar scheduler is out of the scope of this thesis. Its output is the layout engine itself.

We guarantee that the resulting functional attribute grammars are always statically schedulable. Such attribute grammars are compilable to efficient tree traversals (Meyerovich *et al.*, 2013). In contrast with the backtracking search employed by general-purpose constraint solvers, our layout engines perform only value propagations and function applications. The search happens at synthesis (compile) time. Assuming deterministic specifications, our synthesized engines always compute the same layout as general-purpose constraints solvers.

4.3 Modular Synthesis

In this section, we first formalize concepts introduced previously and then present GM synthesis applied to layout engines. To simplify the presentation, we start by explaining our technique on a language containing a single document (*i.e.*, the grammar has a single derivation). In a second step, we generalize our approach to languages of documents and discuss the completeness of GM synthesis (Section 4.4).

Preliminaries

For the sake of readability, we introduce the following notations: Let $f : D^m \rightarrow D^n$ be a function computing n variables given m variables, all in domain D. We denote by $\hat{f}[I, O]$ the function f lifted to symbolic variables, where I is the list of variables read ($|I| = m$), and O is the list of variables computed ($|O| = n$). For example, if $f(x_1, x_2) \overset{\text{def}}{=} (x_1 + x_2, 2x_1 - x_2)$ then $\hat{f}\,[\{a, b\}, \{c, d\}]$ represents $c := a + b$ and $d := 2a - b$. We purposely abstract away the mapping of variables onto arguments. Similarly, for relations of arity $m + n$, for instance $R(x_1, x_2, x_3) \overset{\text{def}}{=} x_1 + x_2 = x_3$, we write $(a, b, c) \in \hat{R}$ to denote $a + b = c$. For convenience, we extend our notation to lists of variables and write $O = \hat{f}(I)$ and $(I \cup O) \in \hat{R}$. In the context of layout, variables range over $\mathbb{Q}$ and are called *attributes*.

Functional Synthesis The functional synthesis problem is to find a total function f given a relation R and a partition of its variables into input/output lists I, O, respectively, such that $(I \cup \hat{f}(I)) \in \hat{R}$ for valuations of I. Such a function exists if R is functional in I. That is, $(I \cup O) \in \hat{R} \land (I \cup \hat{f}(I)) \in \hat{R} \implies O = \hat{f}(I)$ holds. As such, f is semantically unique (but may have multiple implementations). For convenience, we say that *f functionalizes R* with respect to inputs I.

We write $\pi_{I,O}(R)$ to denote the procedure finding such a function; the procedure fails if the function does not exist. GM synthesis relies on a functional synthesizer (π) to perform synthesis locally, on the subproblems created by decomposing the specification. π can be implemented using existing techniques (see Section 4.5).

Blocks and Documents We start with definitions of blocks and documents.

Definition 9 (Block). *A* block *is a pair (V, R), where V is a finite set of attributes and R is a relation over V. Some attributes of V can be marked as* inputs, i.e. *runtime constants. The relation R is the conjunction of the constraints defining the layout semantics of the block. We assume that R is in CNF. That is, R is a conjunction of clauses $cl_0 \land \ldots \land cl_n$.*

Definition 10 (Document). *A* document *is a tree of block-labeled nodes. Each document node contains the attributes and the relation of its block. As such, a block acts as the "type" of a node and through the layout constraints in the relation, the block establishes its layout semantics.*

To represent semantic connections between document nodes, we place additional equality constraints between attributes from a parent and its children (in the tree hierarchy). Formally, a connection c, denoted by $(A, B)_c$, is an equality constraint between the sets of attributes A and B. For now, both A and B are singleton sets.

Finally, given a document d, let I_d be the set of attributes of d marked as input. Let O_d be all other (non-input) attributes of d. Let $rel(d)$ be the relation representing the underlying constraint system of d. Formally, $rel(d)$ is the conjunction of the the relation of every document node as well as the equality constraints stemming from connections between nodes.

Definition 11 (d-Solver)**.** *Given a document d, a d-*solver *is a function $f\ [I_d, O_d]$ which functionalizes $rel(d)$.*

Modular Synthesis To synthesize a d-solver for a particular document, the simplest approach would be to use π directly and compute $\pi_{I_d,O_d}(rel(d))$. This is impractical in practice for all but the most trivial documents, since $rel(d)$ may be large and have more than a thousand of attributes. Consequently, we need a way to divide d-solver synthesis into simpler, independent subproblems. Our approach relies on the following hypothesis: the d-solver can be expressed as a composition of smaller, "local" functions, synthesized from each subproblem individually.

Given a document d, we synthesize a d-solver in three steps: (i) we decompose the specification ($rel(d)$) into conjuncts; (ii) we perform synthesis locally, on each individual conjunct, thus obtaining *local* functions; and (iii) we select and compose just enough local functions to construct a *global* function computing all attributes of d, thus creating a d-solver. Before we detail each of the three steps, we highlight the algorithmic challenges by constructing a d-solver for a small document with the help of an oracle.

Example

Let us consider a document comprising two nodes labeled with block $a \stackrel{\text{def}}{=} (V_a, R_a)$ and block $b \stackrel{\text{def}}{=} (V_b, R_b)$, respectively. The specification of each block is shown below:

$$V_a \stackrel{\text{def}}{=} \{x, y, z, i\} \qquad R_a \stackrel{\text{def}}{=} x = i \wedge i + z = y$$
$$V_b \stackrel{\text{def}}{=} \{x, y\} \qquad R_b \stackrel{\text{def}}{=} x = y$$

Our document has one input, denoted by attribute i. For the sake of the explanation, we abstract away connections. Instead, our two nodes directly share connected attributes. Here, both nodes share attributes x and y. As such, the specification of the document—$rel(d)$—is simply $R_a \wedge R_b$. To create a d-solver, we must synthesize a function computing attributes $O_d = \{x, y, z\}$ from input attribute $I_d = \{i\}$.

Decomposition (Step 1) The first step is to decompose $rel(d)$. We follow the document structure and create two subproblems, one per node of the document.

Local Synthesis (Step 2) The second step consists of generating local functions for each node of the document. First, we ask the oracle to partition each block relation into subsets of clauses. Intuitively, each subset corresponds to one "pass" of the global function through the corresponding block. Then we ask the oracle to partition the attributes of each block into input/output sets. Finally, we synthesize local functions for each of set of clauses using our functional synthesis procedure π. Without the oracle, we would need to enumerate all partitions of clauses, as well as all partitions of attributes.

For our example document, block a is made of two clauses: $x = i$ and $i + z = y$. The oracle partitions R_a into subsets $s_0 \overset{\text{def}}{=} \{x = i\}$ and $s_1 \overset{\text{def}}{=} \{i + z = y\}$. Then the oracle partitions V_a into an input set $I_a \overset{\text{def}}{=} \{i, y\}$ and an output set $O_a \overset{\text{def}}{=} \{x, z\}$. Given these two partitions, we generate local functions for each set of clauses s_0 and s_1 using π. Of course, such functions are not guaranteed to exist. In this case, $\pi_{I_a,O_a}(s_0)$ yields the function $f_1 \overset{\text{def}}{=} x := i$, and $\pi_{I_a,O_a}(s_1)$ produces $f_2 \overset{\text{def}}{=} z := y - i$

We apply the same process on block b. Since R_b is made of a single clause, the oracle trivially partitions R_b into R_b itself. The oracle splits V_b into $I_b \overset{\text{def}}{=} \{x\}$ and $O_b \overset{\text{def}}{=} \{y\}$, then by applying $\pi_{I_b,O_b}(R_b)$, we obtain the function $f_3 \overset{\text{def}}{=} y := x$.

Recomposition (Step 3) The third step consists of constructing a global function functionalizing $rel(d)$ by selecting a subset of local functions and composing them together. This is the key step of GM synthesis.

Since the oracle produced exactly the necessary functions, we now merely need to order them to satisfy their dependencies. That is, for each local function, the attributes read must be computed before the function is applied. We encode function dependencies using a hypergraph whose vertices are attributes and whose edges represent local functions (Figure 4.3.1). The source of each edge indicates the set of attributes read and its destination the set of attributes computed. A topological sort of the hypergraph reveals the order in which to compose local functions. Here, by applying f_1 first, then f_3, and finally f_2, we obtain the desired global function.

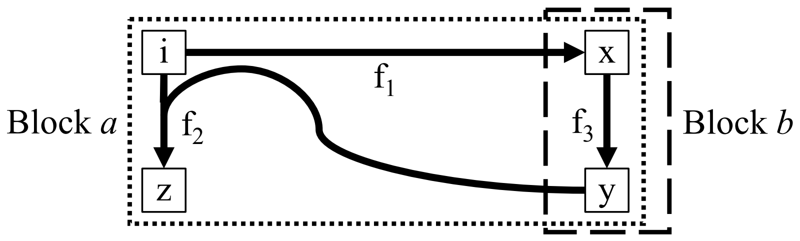


Figure 4.3.1. The hypergraph of the dependencies of f_1, f_2, and f_3. Note that the local function f_2 is represented by a hyperedge with two sources: i and y.

Implementing the Oracle Let's take a step back and analyze the role of the oracle. We relied on the oracle twice during the local synthesis step: the first time to partition block relations into subsets of clauses, and the second time to partition the attributes of each block into input/output sets. Each of these local oracular decisions must be coordinated to achieve global properties not apparent at the local (*i.e.*, block) level:

- *Function Selection* When looking at a block in isolation, we do not know how many local functions are needed to compute all of its attributes. In our example, the attributes of block a are computed with two local functions, in two steps: the value of y is required to compute z, but block b can compute y only if block a has already computed x. If we performed local synthesis directly on block a's relation (R_a), without decomposing it into subsets of clauses, we would be restricting ourselves to solving block a with a single local function, which is not possible in our example.

- *Flow of Computation* While we know the overall (document) inputs, at the block level, we need to determine which attributes are known (inputs) and which attributes will be computed (outputs). The flow of computation is a property of the whole document and is unknown when synthesizing local functions. In fact, the same node may be traversed multiple times by the global function, each time invoking one local function, like the node (labeled) a in our example.

We used the oracle to simplify our synthesis algorithm which *conceptually* relies on global reasoning to synthesize local functions. To gain scalability, we restrict the generation of local functions to block-local reasoning. In the absence of a benevolent oracle, we synthesize local functions considering both all partitions of clauses into subsets and all partitions of attributes into input/output sets. As a result of this exhaustive enumeration, we obtain many more local functions than needed for the construction of the global function. We "implement" the oracle in the recomposition step, in which we must now select which local functions to use. We perform the selection symbolically, by reasoning on a hypergraph summarizing all flows of computation. By selecting local functions, we are indirectly making the same two decisions the oracle made: for each block, we select a clause partition and an input/output partition.

Formalization

We formalize the three steps of GM-synthesis (decomposition, local synthesis, and recomposition) for a language of a single document (Figure 4.3.2). Let d be this document.

Decomposition (Step 1) Conveniently, the structure of the document provides us with an initial decomposition where related constraints are already clustered together by the programmer: we decompose $rel(d)$ at nodes/blocks boundaries.

Note that there is no best granularity of decomposition: it is a trade-off between scalability and completeness of our approach. Finer decompositions lead to smaller relations and hence to more efficient local synthesis, but sometimes small relations are not functionalizable; they need to be conjuncted with other relations to be functional. We discuss completeness of GM synthesis in Section 4.4.

Local Synthesis (Step 2) To start, let us define local functions formally.

Definition 12 (Local Function). *Given a block $(V, R \overset{def}{=} cl_0 \wedge \ldots \wedge cl_n)$, a* local function *is a quadruple (f, I, O, S) where*

1. *I and O are lists of input/output attributes such that $I \subseteq V$, $O \subseteq V$, and $I \cap O = \varnothing$,*
2. *$S \subseteq \{cl_0, \ldots, cl_n\}$ is a subset of clauses,*
3. *f functionalizes S with respect to inputs I: $f = \pi_{I,O}(S)$.*

Note that executing the local function (f, I, O, S) assigns the attributes computed by f with values satisfying all clauses in S.

To generate as many local functions as possible, for each block (V, R) in d, we enumerate both all partitions of clauses of R and all input/output partitions of V, as detailed in Algorithm 4.3.1.

Algorithm 4.3.1. Synthesize local functions for a block.

```
Input: A block $b \overset{def}{=} (V, cl_0 \wedge \ldots \wedge cl_n)$
Output: A set of local functions over attributes $V$
begin
    $R \leftarrow \varnothing$
    foreach subset $S \subseteq \{cl_0, \ldots, cl_n\}$ do
        foreach partition of $V$ into sets $I$ and $O$ do
            if $(f, I, O, S) = \pi_{I,O}(S)$ exists then
                Add $(f, I, O, S)$ to $R$.
        end
    end
    return $R$
end
```

Recomposition (Step 3) We reduce the problem of choosing and composing local functions to finding a particular kind of spanning tree on a hypergraph. The hypergraph encodes a summary of all possible flows of computation between attributes of the document.

Definition 13 (Hypergraph Summary). *Given a document d, an hypergraph summary $H_d \overset{def}{=} (V, E)$ is such that V is the set of attributes of d and E is a set of local functions. Each local function (f, I, O, S) is represented with the hyperedge (I, O), where I is the set of source attributes and O the set of destination attributes.*

Since connections are equality constraints between sets of attributes, we can also represent them with local functions. Recall that, for now, each connection (A, B) is such that A and B are singleton. Let $A \overset{\text{def}}{=} \{a\}$ and $B \overset{\text{def}}{=} \{b\}$. The connection (A, B) is equivalent to $(id, A, B, \{a = b\})$ where id is the identity function.

We construct the hypergraph H_d as follows: For each node n in d labeled with block b, we instantiate the set of local functions of b on the attributes of n. Finally, we add two hyperedges per connection, one for each possible flow of values, either up or down in the document tree. Algorithm 4.3.2 details this process.

Algorithm 4.3.2. Constructing a hypergraph summary encoding all possible compositions of local functions.

```
Input: A document $d$ and a set of connections $C$
Output: A hypergraph summary of $d$

begin
    $E \leftarrow \varnothing$
    foreach node $n$ in $d$ labeled with block $b$ do
        Add $\{(I, O) \mid (f, I, O, S) \in \text{Algo1}(b)\}$ to $E$.
    end
    foreach connection $(A, B)$ in $C$ do
        Add $\{(A, B), (B, A)\}$ to $E$.
    end
    return $(I_d \cup O_d, E)$
end
```

Before we define the d-solver in terms of paths in H_d, let us note the following two facts about the hypergraph summary H_d. First, each hyperpath encodes a function reading its source attributes and computing its destination attributes.

Lemma 4. *Each hyperpath $p = f_0, \ldots, f_n$ in H_d encodes a function $f_p[I_p, O_p] = f_0 \circ \ldots \circ f_n$. Let I_i, O_i be the input/output sets of f_i, the ith function in p. Then $O_p = \bigcup_{0 \leq i \leq n} O_i$ and $I_p = (\bigcup_{0 \leq i \leq n} I_i) \setminus O_p$. From properties of hyperpaths, it follows that:*

1. *The dependencies of each local function on the path are satisfied. For each function f_i with $i > 0$, we have $I_i \subseteq \bigcup_{0 \leq j \leq i-1} O_j \cup I_p$.*

2. *Each attribute is computed at most once: For any pair of functions f_i and f_j in p such that $i \neq j$, we have $O_i \cap O_j = \varnothing$.*

Lemma 5. *Every function f_p defined by a hyperpath p in H_d satisfies the conjunction of clauses of its local functions. Let $p = f_0 \ldots f_n$ be a hyperpath representing function f_p. Let (f_i, I_i, O_i, S_i) be the ith function in p. Then f_p functionalizes $\bigwedge_{0 \leq i \leq n} S_i$. We say that f_p satisfies all clauses* traversed*.*

Lemma 5 follows directly from the fact that, by construction, each local function (f_i, I_i, O_i, S_i) functionalizes S_i. Finally, let us define the subset of paths which can be executed.

Definition 14 (Executable Path). *A hyperpath p in H_d is* executable *iff it starts from the document inputs. That is, the function $f_p[I_p, O_p]$ encoded by p is such that $I_p \subseteq I_d$.*

We are now ready to state under which conditions a hyperpath encodes a d-solver. That is, a global function which functionalizes $rel(d)$ with respect to the document inputs I_d.

Definition 15. *The hyperpath p is an* executable covering spanning tree *iff all of the following three conditions hold: (i) p is executable; (ii) p is a spanning tree; and (iii) p traverses all clauses of $rel(d)$. We call the third condition* coverage*.*

Theorem 1. *Each executable covering spanning p in H_d encodes a global function which functionalizes $rel(d)$ with respect to document input I_d.*

Since p is an executable spanning tree, it follows that both $I_p \subseteq I_d$ and $O_p = O_d$. From the coverage condition and using Lemma 5, we conclude that f_p functionalizes $rel(d)$.

Theorem 2. *If there exists an executable covering spanning tree in H_d, then $rel(d)$ is functional in I_d.*

Since every local function composing the covering spanning tree stems from a functional set of clauses (with respect to local function inputs), one can show that the set of all traversed clauses is functional with respect to I_d. Note that there may be multiple covering spanning trees. Each such tree encodes a semantically equivalent global function, but they may differ syntactically (Figure 4.3.2).

Together, Theorems 1 and 2 show that our approach is correct: the d-solvers synthesized always fulfil the specification. Note that finding a spanning tree in a hypergraph is NP-complete (Warme, 1998). In the next subsection, we explain how to encode the search for a d-solver in SMT after generalizing our approach to languages of documents.

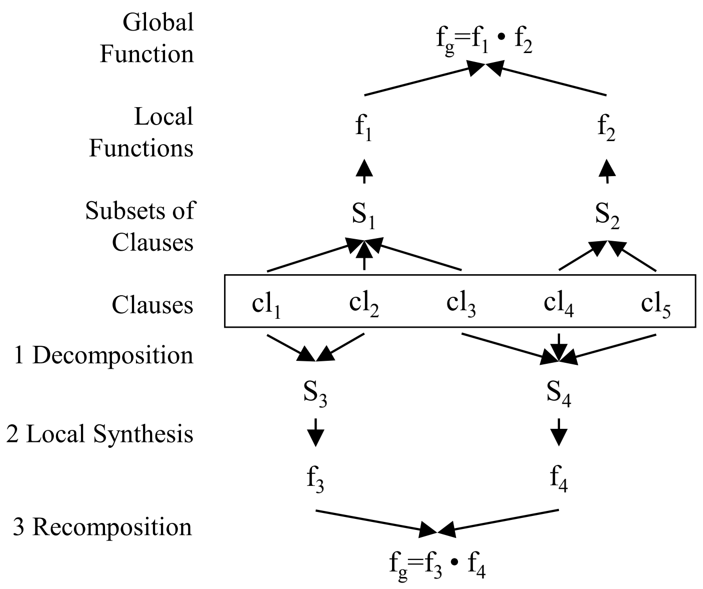


Figure 4.3.2. The three steps of GM synthesis. This diagram shows that two distinct decompositions can lead to syntactically different, yet semantically equivalent, *d*-solvers.

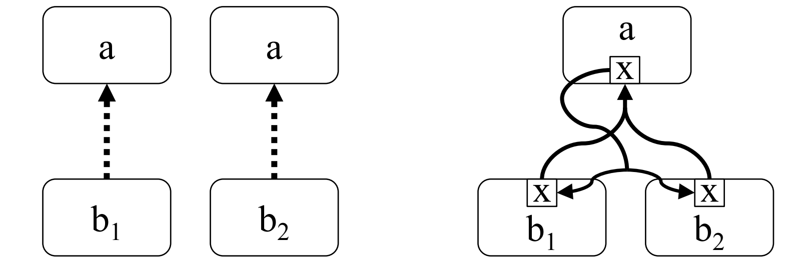
4.4 Grammar-Modular Synthesis

In this section, we generalize the modular synthesis technique presented so far to grammar-modular synthesis for languages of documents. In essence, to support grammars producing more than a single document, we need to handle alternative and recursive productions. By alternatives, we refer to non-terminals having more than one production. We start by formally defining languages of documents and language solvers.

Definition 16 (Language)**.** *A* language of documents *is regular tree grammar $\mathcal{L}$ whose terminals are block-labeled nodes. Each tree in $\mathcal{L}$ forms a document. Each production of $\mathcal{L}$ can place semantic connections between attributes of a parent node and its children.*

Definition 17 ($\mathcal{L}$-Solver)**.** *Given a language $\mathcal{L}$, a $\mathcal{L}$-solver is a statically schedulable functional attribute grammar which defines a d-solver for every document $d \in \mathcal{L}$.*

A language of documents together with blocks definitions form a relational attribute grammar. As a result, we can view the synthesis of a $\mathcal{L}$-solver as converting a relational attribute grammar into a statically schedulable functional attribute grammar. As such, to construct a $\mathcal{L}$-solver, we compute: (i) the mode of all attributes together with a corresponding subset of local functions; and (ii) a total order over attributes. The modes capture whether attributes are inherited or synthesized. The total order prevents cyclic dependencies, which guarantees that the resulting functional attribute grammar is statically schedulable.



(a) The two derivable documents **(b)** Their hypergraph summary

Figure 4.4.1. Encoding alternative productions with hyperpaths. A language of two documents, each stemming from one production of an alternative non-terminal (a), Algorithm 4.3.2 encodes the connection $(\{a.x\}, \{b_1.x, b_2.x\})$ with two hyperedges, thereby enforcing the same flow of computation for both documents (b).

Synthetizing $\mathcal{L}$-Solvers

Given a language $\mathcal{L}$, we synthesize a $\mathcal{L}$-solver as follows: First, we create a *witness* document which exhibits all productions of the grammar of $\mathcal{L}$. Then we create a hypergraph summary of $\mathcal{L}$ by applying Algorithm 4.3.2 on the witness document. Finally, from the hypergraph summary, we construct an SMT formula whose models encode both attribute modes and a subset of local functions. Together, they form a $\mathcal{L}$-solver.

The witness document can be produced easily by unrolling the grammar until every terminal (*i.e.*, node) appears in the document. By doing so, we ensure that $rel(d_w)$ contains all constraints of $\mathcal{L}$.

Connections across alternative productions can be encoded directly as hyperedges with multiple sources or destinations. Conveniently, properties of hyperpaths guarantee that all productions of the same non-terminal will share the same mode (*i.e.*, the same flow of computation). For example, consider the following language where block a may have either block b_1 or b_2 as child. Attribute $a.x$ is connected to either b_1.x or b_2.x.

```
S ::= a(B) with a.x = B.x
B ::= b₁() | b₂()
```

We encode the two alternative productions of the non-terminal B with a single connection c: $(\{a.x\}, \{b_1.x, b_2.x\})_c$. When creating the hypergraph summary, Algorithm 4.3.2 encodes c with two hyper-edges (one with two destinations and one with two sources) representing values flowing either up or down through both derivations (Figure 4.4.1).

To handle recursion, we relax the definition of covering spanning trees (Definition 15) to carefully allow some cycles, those which are created by recursion and do not represent true cyclic dependencies of attributes.

SMT Encoding We encode the existence of a $\mathcal{L}$-solver as an SMT query.

Let d_w be the witness document of $\mathcal{L}$, and let $H_{d_w} = (V, E)$ be its hypergraph summary. Recall that V is the set of all attributes of d_w. Let $F \subseteq E$ be the set of local functions which are not connections, augmented with one function modeling inputs: $(-, \varnothing, I_{d_w}, \varnothing)$. Each local function $(f, I, O, S) \in F$ is encoded with one boolean flag e_f, which is true if f is used in the $\mathcal{L}$-solver; we say that f is *selected*.

We encode each attribute $x \in V$ with two variables:

1. One boolean m_x, representing the mode of x: m_x can either be inherited ($\downarrow$) or synthesized ($\uparrow$).
2. One integer l_x used to impose a total order on all attributes.

We partition the connections of $\mathcal{L}$ two subsets: (i) R, the set of recursive connections, those which stem from recursive nonterminals; and (ii) N, the set of non-recursive connections. Every connection $(A, B)_c \in N \cup R$ is encoded with one boolean m_c representing the mode of the connection: either inherited ($\downarrow$) or synthesized ($\uparrow$).

For each block (V, R) of $\mathcal{L}$, we encode each clause cl of R with one boolean named e_{cl}.

Finally, we define $bmode(x)$, a function converting the grammar mode of attribute x (inherited or synthesized) to a "block" mode (in or out) representing whether x is an input or an output of its block. The block mode is equivalent to modes of logic programs: attributes marked in are computed outside the block and propagated to it through connections; attributes marked out are computed within the block by a local function.

$$bmode(x) := \begin{cases} in & \text{if } \exists (A, B)_c \in N.\ (x \in A \land m_x = \uparrow) \lor (x \in B \land m_x = \downarrow), \\ out & \text{otherwise.} \end{cases}$$

We break our encoding in five parts: (i) connections; (ii) local functions; (iii) the spanning property; (iv) schedulability; and (v) soundness. We explain each of them individually.

Connections The first part encodes the relationship between the mode of a connection and the mode of the attributes connected.

$$\phi_{\text{Conn}}((A, B)_c) := \left(m_c = \downarrow \implies \bigwedge_{x \in B} m_x = \downarrow\right) \quad \land \quad \left(m_c = \uparrow \implies \bigwedge_{x \in A} m_x = \uparrow\right)$$

Functions The second part is divided into two conjuncts: The first conjunct captures the relationship between local functions and attribute modes. Notice that we do not constrain the input of local functions to have an in mode. Doing so would prevent chaining of local functions within the same block, preventing the $\mathcal{L}$-solver from invoking multiple local functions during the same traversal. The second conjunct records all clauses of $rel(d_w)$ traversed by the subset of local functions selected.

$$\phi_{\text{Fun}}(f, I, O, S) := \left(e_f \implies \bigwedge_{x \in O} bmode(x) = out \right) \quad \wedge \quad \bigwedge_{cl \in S} e_{cl}$$

Spanning The third part guarantees that each attribute x is computed by a local function, a non-recursive connection, or inductively by recursion. Note that requiring every attribute to be computed at least once is not sufficient to ensure the soundness of the $\mathcal{L}$-solver. Consider the following grammar with two blocks $a \overset{\text{def}}{=} (\{x\}, x = 2)$ and $b \overset{\text{def}}{=} (\{x\}, x = 1)$:

```
S ::= a(B) with a.x = B.x
B ::= b()
```

Note the connection between the attributes $a.x$ and $b.x$. The only document derivable from this grammar has no solution. However, if we allowed attributes to be computed twice, then we would find a $\mathcal{L}$-solver which first assigns 1 to $b.x$ and then assigns 2 to $b.x$. This example illustrates how the same attribute may be assigned two distinct values, each satisfying one half of the specification. To reject such grammars, we require every attribute to be computed exactly once. As a result, our $\mathcal{L}$-solvers are *single-assignment* attribute grammars, a class of attribute grammars simpler to schedule. We define the logical connective $\odot$ to be true iff exactly one of its clauses is true.

$$\phi_{\text{Span}}(x) := \bigodot \begin{pmatrix} \{e_f \mid (f, I, O, S) \in F \wedge x \in O\} \cup \\ \{m_c = \downarrow \mid (A, B)_c \in N \wedge\ x \in B\} \cup \\ \{m_c = \uparrow \mid (A, B)_c \in N \wedge\ x \in A\} \cup \\ \{m_c = \uparrow \mid (A, B)_c \in R \wedge\ x \in B\} \end{pmatrix}$$

Schedulability The fourth part guarantees the absence of cyclic dependencies by enforcing a total order on attributes. Note, that we only consider the non-recursive connections (N) to allow cycles of attributes caused by grammar recursion. That is, every cyclic path in subgraph of selected local functions must include one recursive connection.

$$\begin{aligned} \phi_{\text{Sched}} := & \bigwedge_{(A,B)_c \in N} \left(m_c = \downarrow \implies \bigwedge_{x \in B} l_x > \max_{y \in A}(l_y) \right) \wedge \\ & \bigwedge_{(A,B)_c \in N} \left(m_c = \uparrow \implies \bigwedge_{x \in A} l_x > \max_{y \in B}(l_y) \right) \wedge \\ & \bigwedge_{(f,I,O,S) \in F} \left(I \supset \varnothing \wedge e_f \implies \bigwedge_{x \in O} l_x > \max_{y \in I}(l_y) \right) \end{aligned}$$

Soundness The final part guarantees that the $\mathcal{L}$-solver functionalizes $rel(d_w)$. We ensure that the local functions selected traverse (cover) all the clauses of (all the blocks of) $rel(d_w)$.

$$\phi_{\text{Sound}} := \bigwedge_{cl \in rel(d_w)} e_{cl}$$

$\mathcal{L}$-Solver Finally, by taking the conjunction of all five parts, we obtain a formula whose models encode both the subset of selected local functions (e_f variables) as well as modes for all attributes (m_x variables) and for all connections (m_c variables).

$$\phi := \bigwedge_{(A,B)_c \in N \cup R} \phi_{\text{Conn}}((A,B)_c) \;\wedge\; \phi_{\text{Sched}} \;\wedge\; \phi_{\text{Sound}} \;\wedge \\ \bigwedge_{(f,I,O,S) \in F} \phi_{\text{Fun}}(f,I,O,S) \;\wedge\; \bigwedge_{x \in V} \phi_{\text{Span}}(x)$$

The translation of models of ϕ to functional attribute grammars is straightforward: The e_f booleans indicate which local functions to use. Note that ϕ also contains a static schedule of the attribute grammar encoded in the l_x variables. In general, our formalism is too abstract to model important execution characteristics like cache locality or parallelization opportunities. We throw away the schedule found and delegate this task to a dedicated attribute grammar scheduler (Meyerovich *et al.*, 2013).

Completeness

GM synthesis is correct (Theorems 1 and 2); the solvers generated are sound: they always satisfy the specification. However, GM synthesis is also incomplete and might fail to find a solver, even when one exists. In Section 4.5, we show that GM synthesis is sufficiently complete in practice.

Recall that GM synthesis relies on the following hypothesis: the global function is expressible as compositions of local functions. The granularity of the decomposition affects whether our hypothesis holds. Coarser initial decompositions (*i.e.*, blocks) yield more local functions at the expense of creating larger local synthesis problems, thus decreasing efficiency. Note that the number of local functions synthesized grows monotonically with the size of blocks only because we consider all subsets of clauses when performing local synthesis.

We call the loss of completeness due to decomposition the *cost of modularity*, to distinguish it from the loss of completeness incurred due to any incompleteness of π. In the next two paragraphs, we state a condition for hierarchical linear systems of equations; this condition is necessary and sufficient to guarantee zero cost of modularity. Finally, we define a class of constraints for which modularity always incurs no cost. For clarity, we state these two properties considering a single document; they are generalizable by induction on the document grammar.

Linear Equations Without loss of generality, we abstract away connections: blocks share connected variables directly, as in the example of Section 4.3. We also assume that the local synthesis procedure π is complete.

Since we are limiting ourselves to linear equations, let the system $rel(d)$ be represented by the matrix of coefficients M_d. The decomposition of $rel(d)$ into blocks corresponds to a partition of the rows of M_d.

Theorem 3 (Completeness Condition)**.** *GM synthesis is complete for linear equations iff M_d can be triangularized (i) using row combinations (*i.e.*, adding a linear combination of rows to another) only between rows belonging to the same block and (ii) using row interchanges for any pair of rows.*

We give an outline of the proof. The first step is to show that the recomposition step of GM synthesis performs the equivalent of back-substitutions on M_d (assuming M_d is upper triangular). For linear equations, local synthesis reduces to row combinations within each block. Indeed, row combinations together with row interchanges form a complete quantifier elimination procedure for linear equations: Gaussian elimination. As such, the power of the local synthesis (π) is exactly row combinations. By requiring M_d to be triangular modulo row interchange after local synthesis, we ensure that $rel(d)$ is solvable with back-substitutions only.

Equality Constraints There exists a (very restricted) class of constraints for which modularity has no cost, regardless of the decomposition: equality constraints. If all atoms of $rel(d)$ are equalities between pairs of attributes, GM synthesis reduces to computing the equivalence classes of $rel(d)$. It is possible to show that equivalences classes are indirectly computed as part of the recomposition step, thus guaranteeing the completeness of modular synthesis.

Of course, equality constraints are too restrictive for all but the most trivial specifications. However, using the same line of reasoning, this result can be extended to demonstrate that the cost of modularity is not affected by the introduction of new equality constraints. As such, simple factorizations of the specification, such as breaking down large constraints into smaller ones have no effect on completeness; a reassuring property for specification authors.

4.5 Evaluation

In this section, we evaluate GM Synthesis along the following three axes:

- *Scalability and Completeness* Since GM synthesis trades completeness for scalability (to a degree controllable with the granularity of decomposition, see Section 4.4), is GM synthesis both scalable and complete enough to synthesize $\mathcal{L}$-solvers for realistic layout languages?

- *Performance* How does the solving speed of our $\mathcal{L}$-solvers compare with state-of-the-art, general-purpose constraint solvers? How do $\mathcal{L}$-solvers and general-purpose constraint solvers scale as document size increases?

- *Parameterizable Layout Engines* Can our layout specifications yield multiple $\mathcal{L}$-solvers, each synthesized for a different set of input attributes, one per user interaction (*e.g.*, resize)? This benefit results from using non-directional constraints which capture flows of values in several directions.

Experimental Setup GM synthesis is parametrized by the local synthesis procedure π. In our experiments, we implemented π with a combination of Sketch (Solar-Lezama *et al.*, 2006) for linear relations and Gröbner Bases (from Mathematica) for polynomial equations. There are many other procedures which could be used to implement π. We note Comfusy (Kuncak *et al.*, 2010) and Mjollnir (Monniaux, 2008).

We used the Superconductor attribute grammar scheduler (Meyerovich *et al.*, 2013) to compile $\mathcal{L}$-solvers to (sequential) tree traversals. The resulting traversals are implemented in JavaScript and operate directly on the browser DOM. As a result, our custom $\mathcal{L}$-solvers can easily be deployed in any web browser. Figures 2.1.3a and 2.1.3b have been laid out by one of our $\mathcal{L}$-solvers.

All our benchmarks were run on a 2.5GHz Intel Sandy Bridge processor with 8Gb of RAM.

Scalability and Completeness

To show that GM synthesis is widely applicable, we demonstrate it on layout languages drawn from the three major layout domains. Our case studies cover: (i) document (webpage) layout; (ii) Graphical User Interface (GUI); and (iii) data visualization. Each of the three languages presented below is full-fledged and computes all attributes needed for rendering.

1. Our first case study is a guillotine layout language where a set of horizontal and vertical dividers partition the space. A subset of CSS can be encoded in such languages (Sinha and Karim, 2013). The guillotine language totals 30 constraints. This is the only language in which all constraints are linear.

2. Our second case study is a language of flexible grids (Feiner, 1988). Such languages are frequently used to layout widgets in graphical user interfaces (Hurst *et al.*, 2009). The sizes of each cell of the grid are allocated based on a weighted sum, producing non-linear constraints. The weight of each cell is a runtime input. The grid language consists of 47 constraints.

3. Finally, a language of treemaps (Johnson and Shneiderman, 1991), a visualization of hierarchical datasets popular in finance. The screen is tiled recursively, based on the area

occupied by each subtree of the document (Figure 1.2.1). Each leaf has a runtime input corresponding to its relative area. Constraints involving area computations are non-linear. The treemap language has 40 constraints.

Our GM synthesizer is sufficiently complete to successfully generate a $\mathcal{L}$-solver for each of the three case studies. The synthesis took less than five minutes, an acceptable compilation time, with the local synthesis step and the recomposition step using approximately equal halves. To illustrate the complexity of the $\mathcal{L}$-solvers obtained after scheduling, Table 4.5.1 lists the number of tree traversals, the number of local functions used, and size of the JavaScript code. For reference, Firefox's layout engine for CSS uses four passes (Atkinson, 2014). Finally, the number of lines of code reported includes only the layout engine itself (*i.e.*, the computation of document attributes); code related to rendering has been explicitly excluded.

We also compare our work with direct functional synthesis techniques, such as Comfusy and Sketch. Such techniques are limited to synthesis of d-solvers, they do not generalize to languages of documents. As such, we apply them on a single small document of 127 nodes. Neither Comfusy nor Sketch could synthesize a d-solver in less than one hour. These results indicate that GM synthesis strikes the right balance between completeness and scalability of synthesis for our domain.

Language	**Tree Traversals**	**Local Functions**		**SLOC**
		Total	Selected	
Guillotine	*td*	289	74	189
Grid	*td* ; *bu* ; *td*	385	89	283
Treemap	*td* ; *bu* ; *td* ; *bu* ; *td*	394	91	341

Table 4.5.1. The complexity of $\mathcal{L}$-solvers for each of our three case studies. The second column shows the number and type of tree passes over the document: *td* denotes a top-down pass and *bu* a bottom-up one. The third columns reports the number of local functions synthesized and the number of local functions used. Finally, the fourth column shows the number of lines of JavaScript code.

Performance

We compare the performance of our synthesized $\mathcal{L}$-solvers with Z3 (De Moura and Bjørner, 2008), a state-of-the-art constraint solver. Note that our solvers are implemented in JavaScript, a relatively slow language. Z3 solves the constraint system defined by the document ($rel(d)$) at runtime. In essence, we measure the ability of GM synthesis to shift the cost of solving constraints from runtime to compile time.

We measured the time to compute the layout of documents from 255 to 16383 nodes, for each of the 3 layout languages outlined above. We argue that such document sizes are typical:

the front page of www.nytimes.com contains over 3000 nodes and data-visualizations tend to be much larger. For each case study, we chose the fastest SMT theory which could express the layout specification. Interestingly, the non-linear arithmetic solver was faster than bivectors for both the grid and treemap languages. For guillotine, we used linear real arithmetic. Table 4.5.2 summarizes our results.

Doc Size	**Guillotine**		**Grid**		**Treemap**	
	GM	Z3	GM	Z3	GM	Z3
255	3	705	5	707	8	680
1023	10	2310	19	1494	49	1935
4095	41	12800	81	8403	120	8935
16383	162	>3 min	213	—	261	—

Table 4.5.2. Time to compute the layout in milliseconds for typical document sizes. Missing entries (—) indicate "unknown" answers (Z3 produced no model). Notice that our $\mathcal{L}$-solvers scale linearly with the document size.

Our $\mathcal{L}$-solvers scale linearly with size of the document, whereas Z3 exhibits exponential behavior on the largest (16383 nodes) document for all three languages. This asymptotic speedup is explained by GM synthesis moving the backtracking-search performed at runtime by Z3 to compile time, leaving only function applications to runtime.

On the medium sized document (1023 nodes), $\mathcal{L}$-solvers are between 39 and 231 times faster than Z3. On the largest document, Z3 was unable to compute a layout within 3 minutes (either timing out or reporting "unknown") for all three case studies. Our results show that across the three case studies, our $\mathcal{L}$-solvers are fast enough (<0.5 second) for interactive settings.

Parameterizable Layout Engines

We demonstrate empirically the expressiveness of non-directional constraints by synthesizing multiple $\mathcal{L}$-solvers from the same specification. Each solver responds to a distinct event or user-interaction by updating the layout. For instance, when the user resizes the main window, one $\mathcal{L}$-solver recomputes the layout using the new width and height as input. We illustrate the power of non-directionality on our language of treemaps.

Imagine a treemap representing the market capitalization of companies. The leaves of the document are companies while inner nodes encode the tiling of the screen (Figure 1.2.1a). Let's consider the following two events: (i) the values of all companies are updated; and (ii) the user resizes the treemap.

Each event defines its own set of runtime inputs from which all remaining attributes are computed. For the first event, the set of runtime inputs is the "value" attribute of each company (*i.e.*, leaf nodes). Given new values, the layout engine must update the sizes of each node, including the overall size of the treemap (root node). In contrast, the second event updates the overall size of the treemap. As such the runtime inputs are the height/width of the root node. The values of leaves remain unchanged, and the layout engine must recompute the scaling parameter converting values (dollars) into areas (squared pixels).

From the same specification, our synthesizer generates two $\mathcal{L}$-solvers, one per set of runtime inputs. For the first event, we obtain (after scheduling) a five pass $\mathcal{L}$-solver, whereas the second event yields a three pass $\mathcal{L}$-solver.

The ability to capture multiple flows of computation within the same specification indicates that relational attribute grammars are a concise formalism for expressing interactive layouts.

4.6 Related Work

GM synthesis builds upon previous work in program synthesis. Our work is closely related to constraint planning, mode inference in attribute grammars, and logic programming.

Program Synthesis

Functional synthesis, a subset of program synthesis (Manna and Waldinger, 1971, 1980), is an instance of the AE-paradigm, also known as the Skolem paradigm for synthesis (Pnueli and Rosner, 1989). GM synthesis builds upon functional synthesis procedures, such as Comfusy (Kuncak *et al.*, 2010) or Sketch (Solar-Lezama *et al.*, 2006), by enabling modular decompositions of specifications to gain scalability.

Constraint Planning (CP)

The task of finding a d-solver can be cast as a multi-way (*i.e.* non-directional) constraint planning problem for which solvers like SkyBlue (Sannella, 1994) and QuickPlan (Vander Zanden, 1996) have been proposed. In CP, each "planning constraint" corresponds to a set of clauses in our framework. Similar to our d-solver setting, given a set of planning constraints, each associated with local functions (methods), a planner finds a sufficient subset of functions that computes all attributes. In contrast with our approach, a programmer is responsible for providing enough local functions as well as partitioning relations, to satisfy special requirements of the algorithm. QuickPlan works in quadratic-time by imposing a clever restriction on planning constraints: each local function must mention all variables of its planning constraint, either as input or as output. The programmer satisfies this restriction by intelligently factoring clauses into planning constraints when writing local functions. In our setting, the same information is left to the oracle (*i.e.*, we search over the space of all factorizations). As illustrated in Section 4.3, our oracle partitions the relation of each block into subsets of clauses, each corresponding to one planning constraint. Without this step, we would be restricted to computing all attributes of each block with a single local function, which would prevent creating layout engines for documents requiring multiple tree passes. In essence, we cannot use QuickPlan to compute d-solvers, because we do not know upfront how many passes are needed. In practice, we synthesize local functions for all subsets of clauses. As a result, we obtain many more local functions than in the traditional constraint planning setting. Naively encapsulating local functions into planning constraints meeting QuickPlan's simplifying assumption would create an exponential explosion. With one planning constraint per subset of clauses, QuickPlan's complexity would become $(2^n)^2$ where n is the number of clauses. In general, constraint planning for non-directional constraints is NP-complete (Maloney, 1992).

We distinguish ourselves by supporting not only finite relations but also tree grammars of relations, enabling the same $\mathcal{L}$-solver to lay out multiple documents (datasets), while still guaranteeing a static schedule.

Attribute Grammar

Our modular synthesis algorithm has close connections with relational attribute grammars and logic programming. Deransart and Maluszynski (1985) give theoretic constructions demonstrating how relational grammars, functional grammars and directed clause programs are related to one another. Mode analysis (Debray and Warren, 1988) techniques for logic programs, which compute whether clause arguments of logical programs are input or output, could be—in principle—transposed to attribute grammars to compute whether attributes are inherited or synthesized. The principal goal of mode inference is to learn static properties enabling compiler optimizations. To this end, such techniques rely on abstract domains to soundly perform over-approximations of modes. Our work differs in two ways. First, to obtain executable $\mathcal{L}$-solvers, we must compute exact modes for all attributes. As such, we cannot apply techniques trading precision for scalability or termination. Secondly, our approach is modular. For each block, we synthesize a set of local functions, which can be viewed as sets of possible modes for a block. Local functions are computed independently for each block and can be reused across layout languages. Mode analysis techniques based on abstract interpretation operate on the whole program.

Constraint Logic Programming (CLP)

In constraint logic programming (Yap, 2004; Apt, 2003; Apt and Wallace, 2007), constraint systems are flat and unstructured while we exploit the tree structure to produce $\mathcal{L}$-solvers in a modular fashion. Furthermore, given a relational specification of a document and a valuation of its inputs, CLP tools search for one layout (*i.e.*, solution) among the potentially many, whereas we ensure that the specification is functional with respect to document inputs. That is, the layout is uniquely determined by inputs (*i.e.*, deterministic).

Chapter 5

Conclusion

Today, new visual layouts are designed and implemented by everyone, from non-technical users to seasoned programmers. Current constraint-based layout languages may be declarative and high-level but still require significant programming knowledge to be used effectively. Prototyping layout remains time-consuming, even for proficient programmers.

This thesis presented a framework for specifying visual layout. By inferring layout specifications from demonstrations and automatically generating layout engines, our framework makes layout programming both easier and more accessible. The framework is divided into three components: a programming methodology (Programming by Manipulation), a specification language (L^3), and compiler-producing executable layout engines. With PBM, users steer the exploration of layout designs by directly displacing blocks of a sample document (Chapter 2). PBM customizes flexible layout blocks, which are specified in L^3. At heart, L^3 phrases layout as a constraint satisfaction problem and abstracts away the flow of computation (Chapter 3). The L^3 compiler is based on a new synthesis algorithm—grammar-modular synthesis—capable of generating tailored layout engines for custom languages of documents (Chapter 4).

Programming by Manipulation has been evaluated by two user studies on both programmers and non-programmers. The first study shows that non-programmers can design interesting visualizations using our PBM tool. The second study demonstrates that proficient programmers are more productive with PBM than with conventional constraint programming. Furthermore, we have implemented grammar-modular synthesis in our L^3 compiler and compared the performance of the resulting engines with state-of-the-art constraint solvers on three layout languages.

This chapter concludes this thesis with a summary of contributions and a discussion of potential directions for future work.

Contributions

Programming by Manipulation addresses 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 potential layouts. We introduce a new type of user demonstration—the What is wrong (WiW) manipulation—which is resistant to users' imprecisions inherent in visual domains such as layout. With such manipulations, users can break constraints and subsequently introduce new ones. Instead of sketching the desired layout, users steer the exploration of designs by pointing out what they would like to change on a given layout. Only the direction of the manipulation is interpreted. Our tool is capable of computing and summarizing ambiguities visually. In our user studies, we have found PBM to be 5-times more productive than direct programming with constraints.

L^3 enables concise and reusable layout specifications using non-directional satisfaction constraints. L^3 guarantees that all constraints are always satisfied, enabling programmers to predictably and reliably control the resulting layout. Unlike most other layout languages, new layout elements can be introduced without stepping out of the language. Non-directionality frees designers from reasoning about artifacts of computation, thus raising the level of abstraction and increasing code reuse.

Grammar-modular synthesis exploits the hierarchical structure of layout specifications to scale synthesis to large relations at the cost of completeness. GM synthesis decomposes specifications into smaller subproblems, which can be tackled in isolation by off-the-shelf synthesis procedures. Our three case studies show not only that GM synthesis scales to large specifications which could not be tackled by state-of-the-art tools, but also that the layout engines generated outperform general-purpose constraint solvers by one order of magnitude. In our experiments, the theoretical incompleteness of GM synthesis did not materialize. We showed that GM synthesis is sufficiently complete to successfully generate layout engines for non-trivial data visualizations, and that our synthesized engines are between 39- to 200-times faster than general-purpose constraint solvers. For our domain, layout, we believe that GM synthesis strikes the right balance between scalability of synthesis, completeness of synthesis, and performance of the resulting layout engines.

Future Work

Participants from our user studies seem interested in a tool combining the manipulator with document authoring so that the sample document can be edited while specifying layout. Our participants were also very enthusiastic about using the manipulator to customize CSS templates. This boils down to expressing CSS with constraints, which has been partially done (Badros *et al.*, 1999). If one could capture all of CSS, a manipulation-based layout system for the web becomes possible, opening PBM to a very large audience.

With PBM, the desired layout is demonstrated on a sample document. As in software testing, a good sample document exercises most of the layout. If the sample document is not representative of the language of documents, undesirable layout behaviors may surprise the user when moving to larger documents. It would be interesting to investigate under which conditions the sample document can be proven sufficient to prevent such situations from occurring.

Another important direction for future work is to study the scalability limits of PBM. The largest tasks of our user studies had slightly over one million configurations. Is there a limit on the size configuration space after which manipulations are ineffective? As the number of configurations grows, is it always possible for users to distinguish layouts with manipulations?

L^3 factorizes recurrent constraints inside traits. A natural question is whether it is possible to decompose the universe of all layout designs into a set of overarching concepts spanning across data visualizations, GUIs, and documents. If so, can we capture each such concept with one trait and create a library of constraints sufficient to express most layouts? Similarly, the limits of satisfiability constraints for layout remain unexplored. For which layouts are richer constraints absolutely required?

Finally, GM synthesis could be applied to domains beyond document layout. For instance, the techniques presented in this paper could potentially generate an attribute grammar-based type-checker from relational type system specifications. The underlying domain of attributes would have to be extended to data types richer than real numbers.

Bibliography

Achugbue, J. O. (1981). On the line breaking problem in text formatting. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 117–122, New York, NY, USA. ACM.

Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA.

Apt, K. R. and Wallace, M. (2007). *Constraint Logic Programming Using Eclipse*. Cambridge University Press, New York, NY, USA.

Atkinson, E. (2014). Personal communication.

Badros, G. J., Borning, A., Marriott, K., and Stuckey, P. (1999). Constraint cascading style sheets for the web. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology*, UIST '99, pages 73–82, New York, NY, USA. ACM.

Badros, G. J., Borning, A., and Stuckey, P. J. (2001a). The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306.

Badros, G. J., Tirtowidjojo, J. J., Marriott, K., Meyer, B., Portnoy, W., and Borning, A. (2001b). A constraint extension to scalable vector graphics. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 489–498, New York, NY, USA. ACM.

Barrett, C., Stump, A., and Tinelli, C. (2010). The Satisfiability Modulo Theories Library. www.smt-lib.org.

Bateman, J., Kleinz, J., Kamps, T., and Reichenberger, K. (2001). Towards constructive text, diagram, and layout generation for information presentation. *Computational Linguistics*, 27(3):409–449.

Borning, A., Lin, R., and Marriott, K. (1997). Constraints for the web. In *Proceedings of the Fifth ACM International Conference on Multimedia*, MULTIMEDIA '97, pages 173–182, New York, NY, USA. ACM.

Borning, A., Lin, R. K.-H., and Marriott, K. (2000). Constraint-based document layout for the web. *Multimedia Systems*, 8(3):177–189.

Bos, B., Çelik, T., Hickson, I., and Lie, H. W. (2011). Css 2.1 spec. w3.org/TR/CSS2/.

Bostock, M., Ogievetsky, V., and Heer, J. (2011). D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309.

Collins, G. E. (1975). Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceedings of the 2Nd GI Conference on Automata Theory and Formal Languages*, pages 134–183, London, UK, UK. Springer-Verlag.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree automata techniques and applications. Available on: www.grappa.univ-lille3.fr/tata.

Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., and Turransky, A., editors (1993). *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg. Springer-Verlag.

Deakin, N., Hickson, I., and Hyatt, D. (2011). Flexible box layout module (w3c working draft). www.w3.org/TR/css3-flexbox/.

Debray, S. K. and Warren, D. S. (1988). Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229.

Deransart, P. and Maluszynski, J. (1985). Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–155.

Di Iorio, A., Furini, L., Vitali, F., Lumley, J., and Wiley, T. (2008). Higher-level layout through topological abstraction. In *Proceedings of the Eighth ACM Symposium on Document Engineering*, DocEng '08, pages 90–99, New York, NY, USA. ACM.

Digia/Qt Project, . (2011). Qml reference documentation (version 4.7). doc.qt.nokia.com/4.7-snapshot/qtquick.html.

Elliott, C., Schechter, G., Yeung, R., and Abi-Ezzi, S. (1994). Tbag: A high level framework for interactive, animated 3d graphics applications. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 421–434, New York, NY, USA. ACM.

Feiner, S. K. (1988). A grid-based approach to automating display layout. In *Proceedings on Graphics Interface '88*, pages 192–197, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

Freeman-Benson, B. N. (1993). Converting an existing user interface to use constraints. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, UIST '93, pages 207–215, New York, NY, USA. ACM.

Hashimoto, O. and Myers, B. A. (1992). Graphical styles for building interfaces by demonstration. In *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology*, UIST '92, pages 117–124, New York, NY, USA. ACM.

Heer, J. and Bostock, M. (2010). Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1149–1156.

Helm, R., Huynh, T., Lassez, C., and Marriot, K. (1992). A linear constraint technology for interactive graphic systems. In *Proceedings of the Conference on Graphics Interface '92*, pages 301–309, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hottelier, T. and Bodik, R. (2014). Program synthesis for hierarchical specifications. Technical Report UCB/EECS-2014-139, EECS Department, University of California, Berkeley.

Hottelier, T., Bodik, R., and Ryokai, K. (2014). Programming by manipulation for layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST' 14, New York, NY, USA. ACM.

Hudson, S. E. and Yeatts, A. K. (1991). Smoothly integrating rule-based techniques into a direct manipulation interface builder. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, UIST '91, pages 145–153, New York, NY, USA. ACM.

Hurst, N., Li, W., and Marriott, K. (2009). Review of automatic document formatting. In *Proceedings of the 9th ACM Symposium on Document Engineering*, DocEng '09, pages 99–108, New York, NY, USA. ACM.

Hurst, N., Marriott, K., and Moulder, P. (2003). Cobweb: A constraint-based web browser. In *Proceedings of the 26th Australasian Computer Science Conference - Volume 16*, ACSC '03, pages 247–254, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Jacobs, C., Li, W., Schrier, E., Bargeron, D., and Salesin, D. (2003). Adaptive grid-based document layout. *ACM Transactions on Graphics (TOG)*, 22(3):838–847.

Johnson, B. and Shneiderman, B. (1991). Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2Nd Conference on Visualization '91*, VIS '91, pages 284–291, Los Alamitos, CA, USA. IEEE Computer Society Press.

Karsenty, S., Landay, J. A., and Weikart, C. (1993). Inferring graphical constraints with rockit. In *Proceedings of the Conference on People and Computers VII*, HCI'92, pages 137–153, New York, NY, USA. Cambridge University Press.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145.

Kuncak, V., Mayer, M., Piskac, R., and Suter, P. (2010). Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 316–329, New York, NY, USA. ACM.

Lau, T. (2009). Why pbd systems fail: Lessons learned for usable ai. *AI Magazine*, pages 65–67.

Lau, T., Domingos, P., and Weld, D. S. (2003). Learning programs from traces using version space algebra. In *Proceedings of the 2nd international conference on Knowledge capture*, K-CAP '03, pages 36–43, New York, NY, USA. ACM.

Lau, T. A., Domingos, P., and Weld, D. S. (2000). Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 527–534, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Lau, T. A. and Weld, D. S. (1999). Programming by demonstration: an inductive learning formulation. In *Proceedings of the 4th international conference on Intelligent user interfaces*, IUI '99, pages 145–152, New York, NY, USA. ACM.

Lie, H. W. and Bos, B. (1997). *Cascading Style Sheets*. Addison Wesley Longman.

Lutteroth, C., Strandh, R., and Weber, G. (2008). Domain specific high-level constraints for user interface layout. *Constraints*, 13(3):307–342.

Maloney, J. H. (1992). *Using Constraints for User Interface Construction*. PhD thesis, University of Washington, Seattle, WA, USA.

Manna, Z. and Waldinger, R. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121.

Manna, Z. and Waldinger, R. J. (1971). Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165.

McDaniel, R. G. and Myers, B. A. (1999). Getting more out of programming-by-demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 442–449, New York, NY, USA. ACM.

Meyerovich, L. (2013). *Parallel Layout Engines: Synthesis and Optimization of Tree Traversals*. PhD thesis, EECS Department, University of California, Berkeley.

Meyerovich, L. A., Torok, M. E., Atkinson, E., and Bodik, R. (2013). Parallel schedule synthesis for attribute grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 187–196, New York, NY, USA. ACM.

Monniaux, D. (2008). A quantifier elimination algorithm for linear real arithmetic. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '08, pages 243–257, Berlin, Heidelberg. Springer-Verlag.

Myers, B. A. (1988). *Creating User Interfaces by Demonstration*. Academic Press Professional, Inc., San Diego, CA, USA.

Myers, B. A. and Buxton, W. (1986). Creating highly-interactive and graphical user interfaces by demonstration. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 249–258, New York, NY, USA. ACM.

Myers, B. A., Giuse, D. A., Dannenberg, R. B., Kosbie, D. S., Pervin, E., Mickish, A., Zanden, B. V., and Marchal, P. (1990). Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85.

Myers, B. A., Zanden, B. V., and Dannenberg, R. B. (1989). Creating graphical interactive application objects by demonstration. In *Proceedings of the 2Nd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, UIST '89, pages 95–104, New York, NY, USA. ACM.

Nevill-Manning, C. (1993). Programming by demonstration. *New Zealand Journal of Computing*, 4:15–24.

Pnueli, A. and Rosner, R. (1989). On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA. ACM.

Sannella, M. (1994). Skyblue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, UIST '94, pages 137–146, New York, NY, USA. ACM.

Schrier, E., Dontcheva, M., Jacobs, C., Wade, G., and Salesin, D. (2008). Adaptive layout for dynamically aggregated documents. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '08, pages 99–108, New York, NY, USA. ACM.

Singh, G., Kok, C. H., and Ngan, T. Y. (1990). Druid: A system for demonstrational rapid user interface development. In *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, UIST '90, pages 167–177, New York, NY, USA. ACM.

Singh, R. and Gulwani, S. (2012a). Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8):740–751.

Singh, R. and Gulwani, S. (2012b). Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 634–651, Berlin, Heidelberg. Springer-Verlag.

Sinha, N. and Karim, R. (2013). Compiling mockups to flexible uis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 312–322, New York, NY, USA. ACM.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. (2006). Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA. ACM.

Souders, S. (2013). How fast are we going now? www.stevesouders.com/blog/2013/05/09/how-fast-are-we-going-now/.

Vander Zanden, B. (1996). An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(1):30–72.

Viegas, F. B., Wattenberg, M., van Ham, F., Kriss, J., and McKeon, M. (2007). Manyeyes: A site for visualization at internet scale. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1121–1128.

Vlissides, J. M. and Tang, S. (1991). A unidraw-based user interface builder. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, UIST '91, pages 201–210, New York, NY, USA. ACM.

Warme, D. M. (1998). *Spanning Trees in Hypergraphs with Applications to Steiner Trees*. PhD thesis, University of Virginia, Charlottesville, VA, USA.

Weitzman, L. and Wittenburg, K. (1993). Relational grammars for interactive design. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 4–11.

Weitzman, L. and Wittenburg, K. (1994). Automatic presentation of multimedia documents using relational grammars. In *Proceedings of the Second ACM International Conference on Multimedia*, MULTIMEDIA '94, pages 443–451, New York, NY, USA. ACM.

Yap, R. H. C. (2004). Constraint processing. *Theory and Practice of Logic Programming*, 4(5-6):755–757.

Zeidler, C., Lutteroth, C., Sturzlinger, W., and Weber, G. (2013). The auckland layout editor: An improved gui layout specification process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 343–352, New York, NY, USA. ACM.

Zeidler, C., Lutteroth, C., and Weber, G. (2012). Constraint solving for beautiful user interfaces: How solving strategies support layout aesthetics. In *Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction*, CHINZ '12, pages 72–79, New York, NY, USA. ACM.