Abstract

We develop algorithms for expressing, verifying and implementing document layout languages. To simplify specification and enable reuse of components, we use relational specifications. Relying on relations, rather than on functions, we free the designer from having to reason about the mechanics of how the layout will be computed. We verify layout languages by proving that all documents in the language will have unambiguous layout. To ease implementation of layout languages, we synthesize a fast propagation layout engine. Our approach relies on recent work for synthesizing functions from relations. We extend the work by making the synthesizer modular, improving scalability when the relational constraint system is naturally modular. We show empirically that layout languages are modular, but our modular synthesizer may be applicable also in other domains. We validate the algorithms on three case studies.

Keywords Program Synthesis, Constraint Solving, Document Layout

1. Introduction

Small domain-specific layout languages for GUIs, magazines, and other data presentations are motivated by unpredictability, complexity, and inflexibility of large layout languages such as CSS. The general purpose layout languages are unpredictable in part because their layout engines are allowed to silently drop document-provided constraints. This is in turn caused by the complexity of undesirable interaction between the features in the language. These languages are also inflexible because, dictated by efficiency, the layout engine fixes a particular propagation strategy which prevents expression of certain layouts.

The layout needs vary across domains. A magazine like New Yorker.com emphasizes perfect text layout while web-mail like gmail is tabular and uses sophisticated scrollboxes. A layout DSL can provide very specific building blocks while avoiding the complexity of a large language. Examples are the predictable GUI languages such as XAML and QML.

We believe that designers would create small layout languages if three problems did not stand in their way: (1) easily reusable components must be created that can be composed into new languages; (2) the designer must ascertain that all documents in a given language are safe, i.e., can be unambiguously solved; (3) an efficient layout engine must be produced. Today, these tasks are laborious.

To express reusable layout components, such as scrollboxes, we use relational constraints. By raising the level of abstraction from functional (directional) specifications to (non-directional) constraints, we allow specification of components that do not commit to a particular layout strategy which is hard-coded when functions are used. We synthesize functional solvers from relations, but the direction of flow depends on the context within which the component is used.

To verify documents, our algorithms ensure that a layout system is safe in that every document in the language will have a unique solution to its layout problem. Proving safety for a language rather than for each individual document allows a syntactic check of document safety: each well-formed document will be solvable.

To construct an efficient layout engine from a declarative specification, we leverage synthesis. From the proof of safety we extract a fast layout engine that is guaranteed not to backtrack; instead of searching, the layout engine solves the document strictly by propagation, with the propagation order being precomputed and represented as an attribute grammar. Our synthesis is modular: by using synthesis at the level of document component, we believe we have struck the right balance between the scalability of the synthesis and the expressiveness of the domain-specific layout languages we can express.

We make the following contributions:
A constraint-based framework for designing, verifying and implementing layout DSLs. The languages are expressed as relational grammars which frees them from having to specify the logic of the layout engine, which we synthesize.

An algorithm for checking whether all documents of a given language can be laid-out unambiguously. A algorithm that synthesizes a suitable layout engine for all document of the language. The synthesized layout engine is a fast propagation solver based on attribute grammars.

Empirical evaluation showing that relational grammars can express custom layout semantics that are not possible in systems where the layout computation is fixed, as in CSS. We also show that our modular synthesis algorithm is scalable and our case studies are easily expressible relationally.

This paper first gives a brief background into document layout. We first address the issue of expressing layouts and synthesizing solvers for single documents in a modular fashion. We then generalize the algorithms to languages of documents. We follow with algorithms for synthesizing propagation functions from relations. We conclude with empirical evaluation.

2. Declarative Layout and Solver Synthesis

After informally introducing the layout problem, this section presents our rationale for (1) the design of layout domain-specific languages and (2) synthesis of their layout engines. Next, we describe the architecture of our system and motivate the algorithms developed in the subsequent sections.

A document is a tree composed of elements such as glyphs, rectangles and images. In our work, each of these elements are represented with variables (also called attributes) and relational constraints over those variables as well as over variables of neighboring element.

A layout language is a set of documents, typically represented as a grammar. The grammar defines syntactically correct trees as well as how constraints connect the elements.

A document distinguishes a set of input variables; these might be the screen size and the sizes of images. The layout problem is to compute the values of document attributes that meet all layout constraints. Typically, the computed attributes are sizes and positions on the screen, as well as many intermediate attributes. In this paper, we abstract away from rendering the graphical skin of each element on the screen; we are only concerned with solving the layout problem.

2.1 Goals and Design Choices

We have made the following design choices.

Declarative layout. We express layouts with relational (non-directional) constraints. In contrast to the the (directional) functional or imperative specification of layouts that is commonly used in practice, constraints have two benefits.

First, they raise the level of abstraction to the declarative level: designers are concerned about the desired properties of the final document rather than about how the layout engine propagates and values during layout computation. Figure 1 illustrates the difference between directional and bi-directional constraints.

Second, constraints are more compact than functions. If multiple events affect the document, a component may require multiple layout functions, each propagating values in distinct directions. For example in a scrollbar component, the motion of the scrollbar changes the position of the content in the view-port, while scrolling the content influences the scrollbar. Currently, programmer write distinct functions for each direction. We synthesize both from the a single relational specification.

Performance Performance of layout engines, especially in browsers, is becoming more critical as the devices shift from desktops to low-power handhelds. CSS designers were careful to ensure that their layouts could be solved in a bounded number of passes of the DOM tree, without any fixed-point computation. To make sure the extra expressiveness that we add does not impair performance, we decided to avoid all search in the solver, for example via back-tracking. This prevents us from using off-the-self tools like Cassowary [4] or Z3 [13]. We synthesize propagation solvers which compute solutions solely by local inference driven by a set of tree traversals. These are the fastest possible solvers short of precomputing the result statically. As a result, we retain the CSS guarantee that all documents will be solved in a bounded number of linear-time passes.

Predicability We believe that one reason why designers find CSS unpredictable is that the CSS layout engine silently drops constraints in order to resolve conflicts in over-constrained documents. We ensure with static analysis that every syntactically correct document in has a (deterministic) unique solution. This property also simplifies the solver, which will not have to handle ambiguity or inconsistent documents at layout time. This property also distinguishes us from existing work on constraint-based layout [4]. Rather
than relying on priority-based constraints (which are satisfied to the extent possible), we rely on the predictable notion of satisfiability.

**Modularity** Reuse is important in layout DSL, which will be built from a library of components. These components are themselves documents; technically, they are either single documents or relational grammars, representing document sub-languages. Because we guarantee a unique solution for each document, we makes it possible to reuse a sub-tree from one document in another document with the assurance that this particular sub-tree will be laid-out the same way. This is not the case with CSS where the unpredictable dropping of constraints might give a radically different layout to the same sub-tree across different documents.

### 2.2 Overview

To simplify the presentation we first explain our methodology on a single document, and then extend it to languages of documents.

**Single document** Our documents are trees of components (also called primitives). Each component has a set of attributes that needs to be computed before the document can be rendered. For instance, they often include positions of each components and their size. Each component is defined by an underlying relation constraining its attributes, which specifies the behavior of the component. The attributes that connect primitives to other primitives are called ports.

Informally, the layout problem is, given a document and some runtime inputs, to compute the all the remaining attributes in the document while satisfying the set of constraints induced by every components specification.

Given a document, we first we ensure the document is safe and then synthesize its layout engine, also called a solver. A document is safe if it is free of inconsistencies and ambiguities. That is, if it has a unique solution for all inputs. If a given document is safe, the propagation solver is extracted from the proof of safety. Finally, given a set of runtime inputs, the solver compute all document attributes from the input values.

In principle, the layout engine for the document can be computed with full functional synthesis [11], which turns a relation with a distinguished set of inputs into a function that computes the remaining variables. In practice, the document induces a relation too large for the algorithm. Therefore, this paper develops a modular version of the relation-to-function synthesis.

The method is divided into three main steps:

1. For each component $P$, we synthesize from its relation $R(P)$ a set of functions $F(P)$. These functions are the available solvers of the component. Each function takes the values of some set of ports of $P$ and computes satisfying values for the remaining variables in $P$. We will construct a document solver by composing these functions.

2. We summarize the document by building an hyper-graph $H$ where each vertex is an attribute and each edge is a potential propagation step. The universe of propagation steps are the primitive-level solvers synthesized in the first step.

3. We use the existence of a spanning in the hyper-graph $H$ as a sufficient condition for safety of the document.

4. If the document is safe, we extract the attribute grammar from the edges in the spanning tree. This attribute grammar is guaranteed to be schedulable into a sequence of tree traversals.

We provide the following guarantees:

- The safe document is deterministic in that each attribute is uniquely defined by the runtime inputs. This means that each document can solved and that the specification is unambiguous, which means that solver implementations will agree on the result of the layout.

- The solver is sound in that for every value of document inputs, the computed values of attributes will satisfy the relational specification of the document.

The solver synthesis algorithm is complete in the sense that if a solver can be composed from primitive-level functions, then we will construct it. However, there are document for which a solver be constructed only when the document is considered as a homogeneous relation, i.e., when the synthesis of relations into functions will not be performed at the level of components. In this case, modular synthesis will, of course, fail to construct a solver.

### 2.3 Layout Languages

The single document approach described in the previous subsection presents two usability challenges:

1. A document is analyzed after it is created, which means that a document designer must be prepared to face error
message pointing out that the document is inconsistent (over-constrained) or ambiguous (under-constrained). In general, document designers are not prepared to reason about constraint satisfaction.

2. A new solver is produced for each document. Every revision of the document must be reanalyzed from scratch. Furthermore, a document-specific solver needs to be shipped with the document, causing document deployment issues. For example, in a web browser setting, a JavaScript evaluator would need to become a part of the document.

To address the first problem, we wish to turn document analysis into a mere syntactic check. If the document is syntactically well formed, it is safe. We believe that syntactic errors are easier to explain than contradictions or ambiguities.

To address the second problem, we wish to create a solver that is applicable for a family of documents. This way, the solver can be shipped once and invoked on any document from the family.

These decisions lead us to layout languages. We will analyze the safety of a grammar; if the grammar is safe, then all documents described by the grammar are safe. If the grammar is safe, we will also synthesize a layout engine, expressed as an attribute grammar, that will work for all documents from the language. Technically, we lift the analyses described above from documents to languages of documents.

We extend our static analysis to handle language of documents. The three main steps remain the same: First we produce a set of propagation functions from the relational specification of every components. Then we select a subset of them to form a spanning tree. Finally we derive an attribute grammar from the tree. The whole process is illustrated in Figure 4. The technical challenge is in the second step, where we need to generalize the hyper-graph representation to work not only for a document but also for a grammar with alternation and recursion.

In terms of usability, we introduce a new role: the layout language designer, who builds a layout DSL. The document writer is now given a language in which every document is safe. He simply builds document inside the language and does not have to deal with ambiguities and inconsistencies anymore. Checking whether a document is in a DSL is simple and can be delegated to the IDE. The solver will work any documents inside a given DSL. This simplifies our deployment problems. For instance, in the case of web-documents, we have a solver per website instead of a solver per page. Figure 3 compares the roles in design of single documents vs. design of layout languages.

**Consequences**  When we generalize from one document to language of documents, we gain flexibility. However, this is not without cost. Some languages will not be safe even if most of the documents one can derive are safe. Thus we loose some expressiveness. The solver maybe slower as it is more generic. However, the solver still made a serie of passes over the tree, which the fastest solver we could envision.

**Claim** In essence, our document languages rely on the following claim, which we believe true: With functional synthesis at the component level, we have struck the right balance between scalability of the analysis and expressiveness of the language.

3. The Single-Document Layout Problem

This section develops the two-stage problem (see Figure 3) where the layout engine is synthesized for an individual document.

3.1 Documents as Modular Relations

In our model, documents are hierarchical, composed of primitives.\(^1\) Primitives are relations that are connected into

\(^1\) In the 1-doc problem, we actually do not need to insist that the doc is hierarchical, i.e. a tree. It can be any composition of primitives but
a document using equality relations on ports of primitives. The document is thus a relation composed of primitive relations. The internal nodes can be thought of as primitives parameterized by their children. Primitives are composable and can be reused across documents.

A document has a distinguished set of variables that act as inputs. The values of inputs are provided at layout time. In a web page, the input may be the size of the screen and the sizes of images. The layout engine computes the values of the remaining variables such that the constraints imposed by the document relation are satisfied.

This is the layout process ends with computing a solution to the constraint system defined by the document relation. In a complete document processing system, primitives would also have a skin, i.e., a graphical representation drawn on the screen, such as the frame and background of a rectangle. In this paper, we are only interested in computing the sizes and screen coordinates of primitives in a document.

To handle multiple kinds of events, a document can have several sets of inputs. For example, a scroll box may have two inputs: the position of the scroll bar and the position of the content in a view-port. The first input is asserted after the scroll bar has been moved, say with a mouse; in response, the position of the content in the view-port is computed. When the content position is asserted, the position of the scroll box is computed. Without loss of generality, we restrict ourselves to a single inputs set in the rest of this paper. In a practical setting, one would synthesize a layout engine for each input set. Note that both engines are computed from the same relational specification. In current approaches, the programmer needs to specify function separately for both directions.

3.2 Preliminaries

Definition 1. A primitive is a pair \((V, R)\) where \(V\) is a finite set of variables and \(R\) is a relation over them. The variables \(V\) are divided into ports and internal variables.

The ports will either be connected with equality constraints to other primitives or will act as inputs.

Definition 2. A document is a triple \((P, C, I)\) where \(P\) is a multiset of (instances of) primitives, \(C\) are equality constraints on ports of primitives in \(P\), and \(I\) a set of ports from \(P\) that are inputs.

As mentioned above, we assume that \(C\) induces a tree structure over \(P\), i.e., the graph whose nodes are \(P\) and edges connect \(p_i\) and \(p_j\) if \(C\) contains an equality constraint over some ports in \(p_i\) and \(p_j\).

3.3 The Layout Problem

A typical layout problem is to compute sizes and positions of document elements. In general, a solution is a valuation of document variables that satisfies all relational constraints:

Definition 3 (Solution to the Layout Problem). Given a document \(D = (P, C, I)\) and a set of initial values for variables in \(I\), a solution to the document \(D\) is a valuation for all variables in \(P\) such that

- all ports in \(I\) are equal to their initial value,
- the underlying relation of each primitive in \(P\) is satisfied,
- the connecting equality relations are satisfied (i.e., all pairs of connected ports are equal).

Definition 4 (Safety). A document is safe if for each valuation of the input ports, there is a unique solution the layout problem.

3.4 Synthesis of a Layout Engine

Given a safe document \(D\), we want to produce a layout engine \(L_D\) that, given an input value \(x\), will compute solution for \(D\). That is, \(L_D(x)\) is a solution to \(D\). In other words, the layout engine is a solver for the constraint system of the document.

This problem of synthesizing a layout engine is a special case of full functional synthesis, a declarative programming problem posed by [11]:

Definition 5 (The Functional Synthesis Problem). Given a relation \(R\) over variables \(V\) and a set of distinguished input variables \(I \subset V\), synthesize a function \(F[R, I]\) that computes the values of the output variables \(O = V \setminus I\) such that \(R\) holds. That is, \(R(I, F[R, I](I))\) holds.

An algorithm to this problem have been presented by [11].

Our modular synthesis follows the structure of the document in that the resulting engine is a composition of functions obtained from the primitives, i.e., each function \(f_i\) is \(F[R(p), I]\) for some primitive \(p \in P\) some input variables \(I \subset Var(p)\). The layout engine computes the value of each output variable with a function from this grammar, where \(v \in I\) and \(f\) is obtained as we just described.

\[ E := v | f(E, \ldots, E) \]

This non-recursive expression yields a propagation layout engine free of search, as desired. Note that \(f\) is a constant-time function whose structure depends on the theory used to express the relation (see Section 5).
The challenge in our modular synthesis is not in synthesizing the functions \( f \) for solving the primitives (this can be done using the algorithm in [11]) but in composing them into a solver for the document. Recall that that while the entire document distinguishes a special set of input variables, the individual primitives do not, even after they are composed into a document. (This is why our documents are relational not functional; the designer is not burdened with specifying the directions of how values flow through the document during the layout.) As a result, the synthesizer needs to discover the directions of these flow. Equivalently, this amounts to determining the distinguished set of inputs for each instance of a primitive in the document.

Since a document does not distinguish a set of inputs for a primitive, we synthesize functions \( f \) for all subsets \( I \) of the variables in a primitive. We then find a composition of these functions such that the composition is a valid solver. We will define a valid solver shortly.

The modular synthesis proceeds in two steps:

1. For each primitive, synthesize functions that compute a solution for the primitive from values of some of its ports.
2. We weave some of these functions together to form a spanning tree that defines a set of tree traversals over the document structure.

### 3.4.1 From Relations to Functions

Let \( R \) be a relation over variables \( V \). For all partitions of \( V \) into an input set \( (I = \{i_1, \ldots, i_n\}) \) and an output set \( (O = \{o_1, \ldots, o_m\}) \) we attempt to find functions \( f_1, \ldots, f_m \) over \( I \) such that \( R(i_1, \ldots, i_n, f_1, \ldots, f_m) \). This is a synthesis problem; we discuss it in more details in Section 5. The functions \( f_1, \ldots, f_m \) are collected to form the primitive summary. This is an hyper-graph whose vertices are ports and whose edges represent propagation functions. Since a function can read multiple variables, an edge can have multiple sources. The algorithm is formalized in Algorithm 1.

### Algorithm 1 Summarize a primitive

**Input:** a primitive with relation \( R \) of the set of ports \( P \)

**Output:** an hyper-graph \( H \) whose set of vertices is \( P \)

\[
H \leftarrow (P, \emptyset)
\]

for all subset \( I = \{i_1, \ldots, i_n\} \subseteq P \) do

\[
O = \{o_1, \ldots, o_m\} \leftarrow P \setminus I
\]

Find functions \( f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n) \) s.t.

\[
\forall i_1, \ldots, i_n, R(i_1, \ldots, i_n, f_1, \ldots, f_m)
\]

if all \( f_1 \ldots f_m \) exists then

\[
H \leftarrow H \cup ((i_1, \ldots, i_n) \rightarrow (o_1, \ldots, o_m))
\]

end if

end for

return \( H \)

### 3.4.2 Weaving Functions Together

This step consists of weaving together some of the functions generated in the previous step to form a spanning tree. Then, we extract from it a set of tree traversals.

The algorithm proceeds as follow: (i) summarize all primitive into hyper-graph; (ii) Connect each summary according to the ports connections; (iii) find a spanning tree in the resulting graph rooted at the inputs. The spanning tree will define a propagation solver because each edge of tree is a function. These functions can then be composed together to form a set of tree traversal. By construction, our solver will not perform any search/backtracking. For unsafe documents, step (iii) will fail: the graph will contain no spanning tree. Thus, we can use the existence of spanning tree as a necessary condition for checking document safety.

A formal description of the algorithm used in step (ii) given Algorithm 2. The function \( freshNode() \) returns a new node, i.e. a node which is not part of the graph.

### Algorithm 2 Merge hyper-graph summaries

**Input:** a set of summaries \( S_1, \ldots, S_n \) and and port connections \( C \)

**Output:** an hyper-graph \( H \)

\[
H \leftarrow \bigcup_{i=1}^{n} S_i
\]

for all pair \( (x, y) \in C \) do

\[
f \leftarrow freshNode()
\]

\[
H \leftarrow addEdge(H, \{x\} \rightarrow \{y\} \cup \{f\})
\]

\[
H \leftarrow addEdge(H, \{y\} \rightarrow \{x\} \cup \{f\})
\]

end for

return \( H \)

**Soundness** The following facts are true about trees in \( H \) rooted at the input set \( I \):

1. All ports which are reachable from the root can be computed by successively applying the functions corresponding to each edge along the path from the root to the port.
2. The presence in a tree of an edge represents the specification of a component guarantees that the specification of the aforementioned component will be satisfied.
3. The presence in a tree of an edge representing the connection between two components guarantees that the equality constraints induced by the connections will hold.

From the first fact, we deduce that we need a spanning tree. A spanning tree guarantees (i) the absence of fix-point computation; (ii) all ports can be computed from the inputs. From the second fact, we see that we need to make sure that there is at least one edge per component in the tree. From the last fact, we conclude that we also need one edge per connection. The last two conditions are the reason for which the fresh nodes were added. They force any spanning tree to go through every component and every connection. Thus,
the existence of a spanning tree in $H$ is a sufficient condition for grammar safety.

**Completeness** Also, let us discuss what we lose by imposing modular synthesis. Each primitive relation will compute its solution in isolation from others (i.e., by using only the ports connected to other primitives). This is a restriction compared to synthesizing a solver for the relation of the entire document. For example, for the relation of some primitive, there may be no way to compute output variables from input variables, no matter how we choose the the distinguished set of input variables. However, if this relation was composed with some other relation, such function may exist.

**Solver Generation** Finally, the construction of a spanning tree in $H$ starting from the input $I$ is matter of performing a depth-first traversal on $H$. Note that, since $H$ contains edges with both multiple sources and multiples destination, the cost of traversing the tree is exponential (in the number of edges) in the worst case.

The last step missing is to generate a solver from the spanning tree. In the single-document case, we can give a function computing the value of each port from the inputs. For every port $p$, it suffices to compose the functions corresponding to the edges along the path from the inputs to $p$.

### 4. Layout Languages

This section generalizes the algorithms presented in the previous section to language of documents.

#### 4.1 3-Stage Architecture

As explained in Section 2.3 and illustrated in Figures 3 and 4, our architecture is divided in three main stages: (1) language design, (2) document design, and (3) execution on the client. In essence, when generalizing from document to language of documents, we have pushed all the static checks and the solver generation to the very first step.

#### 4.2 Document Families Problem Statement

We generalize the layout problem for document families as follows:

**Definition 6.** A language $\mathcal{L} = (P, \mathcal{G})$ is pair made of a set of primitives $P$ and a context-free grammar $\mathcal{G} = (N, \Sigma, R, S)$ where $N \cup \Sigma = P$. Each word of $\mathcal{G}$ is a document, as per Definition 2. The start symbol $S$ defines a set of input ports, which will be the inputs for all documents in $\mathcal{G}$.

Terminals ($\Sigma$) are primitives without children, whereas are non-terminal ($N$) are primitives with children.

**Definition 7 (Language Safety).** A language $\mathcal{L}$ is safe iff all documents in $\mathcal{L}$ are safe.

Our goal is to solve the layout problem (Definition 3) for all documents in $\mathcal{L}$ with a single solver. In other words, we would like to generate one solver per language, not one per document.

#### 4.3 Compilation to Attribute Grammar

Recall our decision to represent the layout engine as an attribute grammar. This choice guarantees that the solver will be a propagation solver, i.e., it will be free of search/backtracking and will be computed with a constant number of traversals of the document tree.

One can view the single-document case as a language which has a single document. Such a grammar would have no recursion and no alternation. These are two new concepts which separates the single-document case from the language families case. We describe our algorithm in details in the next section. As in the single-document case, we will proceed by a reduction to finding a spanning tree in a hypergraph (c.f. [8]).

##### 4.3.1 Reduction to Hyper-Graph

First, we summarize each primitive into an hyper-graph using Algorithm 1. However, merging summaries to produce a single graph representing the whole grammar becomes more complicated. In a single (finite) graph, we need to capture a potentially unbounded number of document. We describe below how we proceed.

We refer to applying Algorithm 1 on a primitive $X$ with the notation $S(X)$. Furthermore, we will use $cnct(p, X)$, where $p$ is a port and $X$ a set of primitives, to denote the set of ports of primitive in $X$ with whom $p$ is connected. W.l.o.g., we will assume that right-hand side of every rule in the grammar has the form $\xi_1 \mid \ldots \mid \xi_n$, where each $\xi_i$ has the form $A_1 \cdot \ldots \cdot A_m$. Algorithm 3 formalizes our approach to merging summaries.

**Algorithm 3 Compute hyper-graph summary for a document language**

| Input: a language $\mathcal{L} = (P, \mathcal{G})$ |
| Output: an hyper-graph $H$ summarizing $\mathcal{L}$ |
| $H \leftarrow \bigsqcup_{X \in P} S(X)$ |
| for all rules $R$ of the form $(X ::= \xi_1 \mid \ldots \mid \xi_n) \in \mathcal{G}$ do |
| for all ports $p \in X$ do |
| $Q \leftarrow cnct(P, \bigcup_{i=1}^{n} \xi_i)$ |
| if $Q \neq \emptyset$ then |
| $r \leftarrow freshNode()$ |
| $H \leftarrow addEdge(H, p \rightarrow (Q \cup \{f\})$ |
| $H \leftarrow addEdge(H, Q \rightarrow \{p\} \cup \{f\})$ |
| end if |
| end for |
| end for |
| return $H$ |

**Alternation** To be sound, we need to make sure that when all the components which are part of an alternative have same set of inputs/outputs ports. If that is the case, in all
safe documents they can be swapped for one another and the document stays safe. To achieve this, we use hyper-edges to enforce structural constraints on the alternations.

**Recursion** Algorithm 3 does not handle recursion. Without recursion, our layout languages can only contain a bounded number of documents. We discuss below how our implementation deals with recursion.

We check the safety of of language by structural induction on the grammar. W.l.o.g. we only consider self-recursive rules. First we remove all self-recursive non-terminals from the right-hand side of each rule in the grammar, and we run Algorithm 3 on the new grammar to construct $H$. Then we find for all the spanning trees in $H$. Note that, by construction of the graph, the base case hold for all spanning trees. The second and last step consists of checking the inductive case on each spanning tree one by one. Let the interface between two components be which connections flow bottom-up and which flow top-down. We require that the interface between a self recursive non-terminal and its parent to be same as the interface between the same non-terminal and any of its children. This is formally stated in Algorithm 4.

Algorithm 4 Check inductive case

**Input:** a language $L = (P, G)$ and a spanning tree $T$

**Output:** True if $L$ is safe, False otherwise

for all self recursive rule $(X ::= \xi_1 \mid \ldots \mid \xi_n)$ do
    for all $P$ such that $P$ produces $X$ in $G$ do
        for all $C$ such that $X$ produces $C$ in $G$ do
            if $\exists x \in P, y \in X, z \in C$ s.t.
                $(x \rightarrow y \in T \land z \rightarrow y \in T) \lor$
                $(y \rightarrow x \in T \land y \rightarrow z \in T)$ then
                return False
            end if
        end for
    end for
end for
return True

This approach has the inconvenience of requiring the computation of all spanning trees in a hyper-graph: since we verify the inductive case in a second pass, the first spanning we find may not be inductively safe, but there might exists another spanning tree which is. Finding all spanning trees is expensive.

Is it possible to encode the base case and the inductive case of a recursive rule into a single hyper-graph? With such an encoding, it would be sufficient to find a single spanning tree. We are investigating this question.

4.3.2 Producing an Attribute Grammar

This step is simple. The spanning tree found with Algorithm 3 defines a subset of propagation functions and connections. This exactly the subset which should be included in an attribute grammar so that (i) it solves the layout problem; (ii) it has a topological sort. The attribute grammar becomes simply the production rules of the relational grammar together with the assignment induced by the edges of the spanning tree.

5. Synthesizing Functions from Relations

We have taken advantage of recent advances in synthesis and used Sketching as well as Comfusy to solve this synthesis problem when logic of $R$ is decidable. We have also used the symbolic reasoning built into Mathematica when dealing with non-linear constraints. In practice, the combination of these techniques was able to synthesize all functions for all the layout system we have modeled.

When a relational specification contains independent conjuncts, i.e. conjuncts which do not have any shared variable, we treat them separately. For each conjunct $c$, we pick an appropriate synthesis procedure by looking at the logic of $c$, then we apply Algorithm 1 on it.

We have used the following synthesis techniques:

5.1 Deductive Synthesis (Comfusy)

Comfusy [11] takes a $R$ relation over variables $V$ as well as a partition of $V$ into inputs/outputs and produces functions that compute the outputs from the inputs. Comfusy is both sound and complete. It relies on quantifier elimination procedures, and is applicable on any logic that admits quantifier-elimination. In our experiments, we used Presburger arithmetic as well as linear rational arithmetic for our layout problem. The scalability of Comfusy depends directly on the complexity of quantifier-elimination in the logic used.

5.2 Inductive Synthesis (CEGIS)

The counter-example guided inductive synthesis (CEGIS [18]) approach relies on a template of the function to be synthesized. The semi-algorithm alternates between instantiating a template and verifying it. If the verification fails, the counter-example is used to create a refined instantiation of the template. This process continues until the verification step succeeds. As opposed the deductive synthesis, inductive synthesis requires a syntactic description of the space of functions to produce templates. Furthermore, termination is not guaranteed when there exists no function satisfying the relation. We used of-the-shelf SMT-solvers ([13], [6]) in our implementation.

5.3 Other Techniques

The two previous techniques are unable to deal with non-linear relations. In our experiments, we have encountered a few primitives which were non-linear. We have noticed that in all those cases the relation had the form of a (conjunctive) system of equations. This made possible the use of Gaussian elimination and Gröbner basis computation. Note that when the relation itself is non-linear, the system of equations sometimes becomes linear once the inputs are fixed. For this
cases, our tool leverages the symbolic reasoning facilities of Mathematica [20].

6. Experiments
Using three layout DSLs, this section evaluates the following hypotheses:

- **Synthesizer performance.** The modular synthesizer based on computation of spanning trees in a hyper-graph is sufficiently efficient. The algorithm has exponential time complexity, so its running time is a potential practical concern.

  We found that even the less efficient algorithm from Section 4 finds a solver in less than 30 seconds.

- **Expressiveness.** Does the ability to formulate a layout DSL and synthesize a custom layout engine enable layouts not possible in fixed layout languages like CSS?

  Two of our layout languages express layouts not possible in CSS.

- **Declarative layouts.** Developing layout languages with relational grammars frees the developer from the concerns of how the layout is computed. Ideally, the developer need not think in terms of primitive propagation functions and the direction of flow during layout.

  We found that declarative reasoning was the case for two of the three languages. The nature of third language required an iterative solver and we had to break the cycle with a little operational reasoning.

- **Restrictions imposed by modular synthesis.** A layout engine can be synthesized in a modular fashion only if individual primitives can be transformed into functions and those functions can be composed into a solver. A typical layout languages modular or do they instead require a whole-document approach?

  We found our languages to be modular.

6.1 Implementation and Synthesizer Performance
We have implemented a prototype which performs the analysis described in Sections 3 and 4. The implementation follows the first (2-phase) algorithm from Section 4. The algorithms have been implemented in Scala. For all three case studies presented in the rest of the section, generating the proof of safety of the grammar and deriving a solver took less than 30 seconds on a computer with a 2.8 GHz Intel Core 2 CPU. For synthesis, we mostly relied on Sketching and Mathematica.

6.2 A Language of Bounding Boxes and Scrollboxes
The first language tests our ability to specify (1) box layouts similar to those in GUI languages like the Mozilla XUL. To stress test composition, we add into the (2) language of scrollboxes, which are interesting in their own right because they demonstrate how the two distinct layout solvers needed for a scrollbox can be synthesized from a single relational specification.

This language uses four kinds of components:

- **Vertical/Horizontal Divider:** Partition the available space between its children vertically or horizontally. These are the HDiv and VDiv components.

- **Vertical/Horizontal Grouper:** Group children together vertically or horizontally by creating a box just large enough to encompass them. These are the HGroup and VGroup components.

- **Scroll Box:** Use x- and y-sliders to display a bigger content box inside a smaller view-port box. The component name is ScrollBox.

- **Leaves:** We have two kind of leaves: Those with a fixed hard-coded size are called Box, whereas the unconstrained leaves are named Leaf.

**Relational Specifications of Components** We have chosen to give the specification of the horizontal divider and the scrollbox. Remaining components are similar. Notice how conciseness of the relational specification: only the behavior is specified; nothing in the constraints describes the actual computations necessary to solve the layout problem. The HDiv primitive is specified as follows:

\[
\begin{align*}
\text{width} &= \text{c}_1 \text{width} + \text{c}_2 \text{width} \\
\text{c}_1 \text{width} &= \text{c}_2 \text{width} \\
\text{height} &= \text{c}_1 \text{height} = \text{c}_2 \text{height}
\end{align*}
\]

As a convention, we prefix ports connected to children with “c.” We give components with exactly two children. In the system, desugaring will turn lists of children into a tree of such two-children components.

![Figure 5. The meaning of the Scroll Box ports. Only the vertical dimension is shown here.](image)
A simplified ScrollBox is defined by the following equations:

- `viewport_vpos + slider_vpos = height * c_height \land
viewport_hpos + slider_hpos = width * c_width`

For the meaning of each port, please refer to Figure 5. In this case, the specification is non-linear. We used the Mathematica symbolic engine to synthesize the propagation functions. We synthesize four propagation functions: Each port can be computed from the three others.

**Relational Grammar** The grammar describes (1) syntactically legal documents, (2) input ports for these documents, and (3) the relations that connect ports of components in the document:

- **Inputs:** `S.height, S.width`
- `ScrollBox1.viewport_{v,h}pos, ScrollBox2.viewport_{v,h}pos`
- **HDiv**
- **HDiv**
- **VDiv**
- **HGroup**
- **VGroup**

Note that the symbol “·” stands for concatenation. For the sake of readability, the equality relations describing the connections between components have been omitted. Instead, we show in Figure 6 these relations for a fragment of the grammar, specifically how HDiv connects to its children. (The unconnected ports in ScrollBox are internal variables used for rendering.)

**Synthesized Solver** The language has been shown safe and the synthesized solver lays out documents in this language in two passes: a top-down pass followed by a bottom-up pass. We present below a fragment of the attribute grammar generated by our tool. (The vertical dimension has been omitted because of lack of space.) An attribute compiler turns this attribute grammar into visitor functions that perform the two traversals.

- `HDiv ::= (VDiv | ScrollBox1 | Leaf1)
  \cdot (VDiv | ScrollBox2 | Leaf2)
  HDiv.c1_width := HDiv.width/2
  HDiv.c2_width := HDiv.width/2
  VDiv1.width := HDiv.c1_height
  ScrollBox1.width := HDiv.c1_height
  Leaf1.width := HDiv.c1_height
  VDiv2.width := HDiv.c2_height
  ScrollBox2.width := HDiv.c2_width
  Leaf2.width := HDiv.c2_width
  ScrollBox.c.width := HGroup.width`

**Bi-directionality in the scrollbox** The layout of scrollbox can have two different inputs: (i) the position of the slider (i.e., scrollbar) and (ii) the position of the content in the view-port. Depending on the event (e.g., the mouse moves the slider vs. the page-down key moves the content), one or the other is set. The above relational grammar assumes that the latter input. After changing inputs in the grammar to `slider_pos`, the synthesizer will confirm that the language is safe even for this input and will produce a layout solver that will be invoked after the slider event. We have used one relational specification to synthesized solvers for multiple events.

We believe that writing the relational grammar and the components specification is simpler that writing the full attribute grammar computing the layout. The relational grammar only specified the behavior, the flow of computation is inferred by our tool; different flow may be needed for each event.

**6.3 Beyond CSS**

One of the key limitation of CSS is that the width of every DOM node must be computed before its height. This makes it impossible to derive the width of an element from its height. It turns out that even if computing width first is suitable most of the time, there are cases when it is useful to determine the height first. Here is such a use-case: Let’s take a document which has on the right a sidebar with frequently used shortcuts. The main content, which can be long, is displayed in the left pane. Since the right side contains shortcuts, it would be nice to make sure they all fit in a single screen, so that there is no need to scroll. Thus, the designer would like the right pane to be just wide enough avoid the need for a scroll box. We can easily capture his intent with
the following constraints.

\[
\begin{align*}
\text{width} &= c_1 \text{width} + c_2 \text{width} \land \\
L_{\text{height}} &= c_1 \text{height} \land \\
R_{\text{height}} &= c_2 \text{height}
\end{align*}
\]

where the ports prefixed by \(c_1\) and \(c_2\) are connected to the children representing the left and right pane respectively. Here, \(\text{width}\) and \(L_{\text{height}}\) are input.

Our tool produces the following attribute grammar (connections are omitted):

- **SideBar**: \(LP_{\text{ane}} \cdot RP_{\text{ane}}\)
  - \(c_{1,\text{height}} := L_{\text{height}}\)
  - \(c_{2,\text{width}} := \text{width} - c_{1,\text{width}}\)
  - \(R_{\text{height}} := c_{2,\text{height}}\)
- **LP_{\text{ane}}**: \(\ldots\)
- **RP_{\text{ane}}**: \(\ldots\)

Once scheduled, this grammar leads to the following computation: The left pane is processed first by computing the width of its element depending on their height. Then, the right pane is solved by computing height from the width.

### 6.4 Adaptive Multi-column Layout

This challenge problem was posed to us by Jim Larus. The layout problem is organize text into columns whose number adapts to the width of the display; a narrower page will have fewer columns. Column heights are to be about equal. This problem cannot be solved in CSS2, although approximations involving hackery with floating page elements have been developed. The demand for the multi-column layout resulted in adding multi-column layout into HTML5 but few browsers implement it as yet.

In this case study, we wish to argue that custom layout semantics such as that of multi-column perhaps need not be hard-coded into the web browser. Instead, we show that a designer can use relational constraints to define the layout concisely and from first principles; we then synthesize the layout solver. We illustrate the process on our our adaptive multi-column component.

In this paper, we have abstracted away from the layout of text. The delicate issues of font kerning and such are best left to font libraries that should be reused in higher-level layout languages. For multicolumn layout, we assume the existence of a function \texttt{TextTool} that lays out given text in a specified width and returns the height that the text will occupy under that width. We will model this library function as a black-box constraint component, which will be terminal in the relational grammar. As opposed to other components, it has no visual appearance and its code is given rather than synthesized from a relational specification. The invocation of such black-box illustrates how we connect to external constraints represented as functions.

The constraints of the multi-column components must first relate the number of columns, the text and the page width. We make each column of equal height and set it to an equal proportion of total text. (We actually need to add some slack to each column, but let us ignore that issue for the sake of simplicity.)

\[
\begin{align*}
\text{num}_{\text{col}} &= \min(\text{width}/\text{col}_{\text{width}}, 2) \land \\
\text{text}_{\text{height}} &= \text{TextTool} \land \\
\text{text}_{\text{height}} &= c_1 \text{height} \cdot \text{num}_{\text{col}} \land \\
c_2 \text{height} &= c_1 \text{height} = \text{height}
\end{align*}
\]

The component has a fixed number of columns (our current system does not allow dynamic creation of elements). So next we hide the columns that are not used. A column with zero width will not be displayed.

\[
\begin{align*}
0 < \text{num}_{\text{col}} &\leq 1 \implies (c_{1,\text{width}} = \text{col}_{\text{width}} \land \\
&c_{2,\text{width}} = 0) \land \\
1 < \text{num}_{\text{col}} &\implies (c_{1,\text{width}} = \text{col}_{\text{width}} \land \\
&c_{2,\text{width}} = \text{col}_{\text{width}})
\end{align*}
\]

We create our multi-column component as a composition of two textboxes, one for each column, and the black-box. The following grammar shows the composition:

**Inputs:** \(\text{MultiCol}_{\text{width}}, \text{MultiCol}_{\text{col}_{\text{width}}}, \text{MultiCol}_{\text{text}}\)

**MultiCol**: \(\begin{align*}
\text{Col}_{(1,2),\text{height}} &= \text{MultiCol}_{\text{c}_{(1,2),\text{height}}} \\
\text{Col}_{(1,2),\text{width}} &= \text{MultiCol}_{\text{c}_{(1,2),\text{width}}} \\
\text{Col}_{(1,2),\text{text}} &= \text{MultiCol}_{\text{text}} \\
\text{TextTool}_{\text{width}} &= \text{MultiCol}_{\text{col}_{\text{width}}} \\
\text{TextTool}_{\text{height}} &= \text{MultiCol}_{\text{text}_{\text{height}}} \\
\text{TextTool}_{\text{text}} &= \text{MultiCol}_{\text{text}}
\end{align*}\)**

Here we have only considered textboxes as the content of each column. This could be extended to a more general layout language, like our bounding box language.

**Synthesized solver** The solver we generate performs the following steps: (1) compute the number of columns; (2) compute the text height by going through the \texttt{TextTool} child; (3) compute the height of each column and pass it down to each column box; (4) report the overall height the multi-column component the parent box.

### 7. Related Work

**Layout with Constraints** Document layout has been specified with constraints in prior work [7, 9, 12, 14–16, 19]. The work most closely related to us is the use of hierarchical constraints for specifying semantics of layout languages [3–5]. In hierarchical constraint languages, constraints are assigned priorities and the solver attempts to maximize the utility value of constraints satisfied. Our approach is based on satisfiability, i.e., all constraints must be satisfied. We believe that silently not satisfying some constraints leads to unpredictabilities. We also differ in the style of the solver. While Cassowary [4] is an incremental Simplex solver, we
use a simpler and more efficient solver based on attribute grammars, that is guaranteed to perform only propagation. We can argue that our is the fastest solver possible since its work is completely devoted to computing the solution rather than discovering how it should be computed. Adaptive layouts have been supported with (directional) functional layout languages [10, 17]. These papers use functions to achieve predictability (avoiding constraint dropping). In contrast, we strive to achieve predictability with satisfiability and safety analysis.

**Constraint Logic Programming (CLP)** There is a huge body of work on CLP ([21], [1], [2]). In the constraint programming setting, constraint systems are flat, unstructured. Our approach exploits the tree structure of our constraint systems to make synthesis modular. Furthermore, given a constraint system, CLP tools search for one solution, whereas we ensure that all solutions are uniquely determined by a distinguished set of variables (i.e. the inputs), and produce a solver which, given values for inputs, computes the unique solution. Moreover, CLP typically works with variables belonging to finite domains. Even if screens are bounded surfaces, we chose to specify our components with variables over unbounded domains by using rationals, to show that our modular synthesis approach is not restricted to layout problems.

8. Conclusion

We have addressed the problem of declarative specification of domain-specific layout languages. We have developed algorithms for (1) modular verification of safety of these languages and (2) modular synthesis of layout engines for documents from these languages. The algorithms have been presented in the context of layout DSLs but their applicability extends to modular synthesis of functions from modular relations. We have found the algorithms efficient and the declarative approach expressive and flexible. Please refer to the beginning of Section 6 for a more detailed summary of empirical evaluation.

References


