Synthesis of Layout Engines from Relational Constraints

Thibaud Hottelier
Ras Bodik

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-181

http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-181.html

November 19, 2014

Synthesis of Layout Engines from Relational Constraints

Thibaud Hottelier
UC Berkeley
tbh@cs.berkeley.edu

Ras Bodik
UC Berkeley
bodik@cs.berkeley.edu

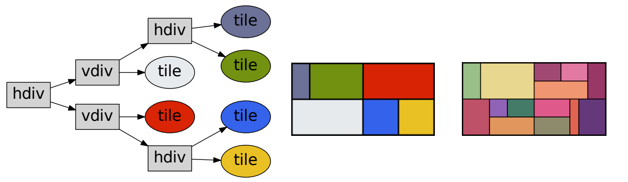
Abstract

We present an algorithm for synthesizing efficient document layout engines from relational specifications. These specifications are high level in that a single specification can produce engines for distinct layout situations. Specifically, our engines are functional attribute grammars, while the specifications are relational attribute grammars. By synthesizing functions from relations (constraints), we obviate the need for constraint solving at runtime, shifting this cost to compilation time. Intuitively, the synthesized functions execute only value propagations and bypass the backtracking search performed by constraint solvers. By working on hierarchical, grammar-induced specifications, we make synthesis applicable to previously intractable relational specifications. We decompose them into smaller subproblems which are tackled in isolation by off-the-shelf synthesis procedures. The functions thus generated are subsequently composed into an attribute grammar which satisfies the relational specification. Our experiments show that we can 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. Introduction

Visual layout is the process of arranging visual elements such as paragraphs or images. A layout is computed by fixing the sizes and positions of some visual elements and using layout constraints to compute sizes and positions of remaining elements. The elements are organized in a *document tree*; the layout constraints are usually local and are attached to document nodes. For example, in Figure 1a, *hdiv* computes its width as the sum of its children widths.

Modern layout languages (*e.g.*, CSS) are loosely based on *functional* attribute grammars (AG) [**?**] which define both syntactically legal documents and their layout semantics. Functional AGs are attractive because they can be solved efficiently using a *layout engine*, an evaluator with a fixed set of document tree traversals. However, the performance gained thanks to a fixed computation strategy comes at the cost of limited expressiveness. For instance, CSS always computes width before height. As a result, CSS cannot express layouts such as treemaps (Figure 1) where *vdiv* derives width as a function of height.

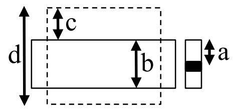


(a) Document (b) Layout of (a) (c) A larger treemap

Figure 1. A treemap visualization. A document tree (a), together with its layout (b). Figure (c) shows the layout of a larger and deeper document from the same layout language.

We sidestep the limitations of (directional) functional constraints by specifying layout with (non-directional) relational constraints [**?**]. Relational constraints do not fix the computation strategy: the value "flows" between variables depending on which values happen to be known. As an example, consider a scroll box containing (i) a textbox (dashed rectangle), part of which is visible in a view port (solid rectangle); and (ii) and a scroll bar, which indicates the position of the textbox relative to the view port. The layout of the scroll box is specified with this relational constraint:

$$\frac{a}{b} = \frac{c}{d-b}$$



When the user moves the textbox, she fixes the textbox position c, necessitating recomputation of the scroll bar position given that we know the value of c. Conversely, when she moves the scroll bar, the textbox position is recomputed from the new value of a. Each such interaction triggers a different flow of computation, but maintains the same constraints.

However, efficient solving of constraint-based layouts remains a challenge. While the layout of a document can be computed with a general-purpose constraint solver, such solvers are up to 200-times slower than tree-traversal engines (see Section 5), which is insufficient for interactive settings, especially on slower, mobile devices.

We present a synthesis algorithm for translating relational layout specifications into efficient (*i.e.*, statically schedulable) functional attribute grammars. For simplicity, we illustrate synthesis on the flat constraint system of the scroll box. Our algorithm derives directional functional constraints for updating the document position ($c \leftarrow a(d-b)/b$) and for updating the scroll bar position ($a \leftarrow cb/(d-b)$). The key point is that both functional constraints can be derived from the single relational constraint. Relational constraints thus offer non-redundant encoding and freedom from a fixed computation strategy.

Single-Document Synthesis The most efficient layout engine is one created for a specific single document. The layout specification of a document is the conjunction of local relations from all tree nodes. The Comfusy algorithm [?] can translate such specifications to functions. This function would then solve the layout of the document. Scaling synthesis to large specifications is a challenge, however. So far, synthesis has been mostly limited to producing program fragments: Our experiments show that Comfusy scales to relations of up to 100 variables. However, layout specifications can include 10^4 program variables per document.

Modular Synthesis To scale synthesis to large specifications, we rely on the hierarchical structure of the specification, specifically a conjunction of smaller relations. In the case of a single document, each node carries a *block* of constraints. We decompose the synthesis problem at node boundaries, into smaller subproblems whose solutions (local functions) are composed to form a global function that implements the overall relation. We trade completeness for efficiency: modular synthesis cannot perform deduction across decomposition boundaries (only function composition), so we may fail to produce a global function when one exists. The reason is that smaller relations may not be functional, preventing us from producing the necessary local functions.

Grammar-Modular Synthesis So far we have outlined how modular synthesis can generate functions solving one particular relation, or equivalently one document. To be practically useful, we must synthesize a single layout engine generic enough to solve any document from a language of documents. We specify such languages using grammars of trees. Figures 1b and 1c show the layouts of two documents from a language of treemaps computed by the same synthesized engine. In essence, we generalize modular synthesis to accept not a fixed relation but a language of relational specifications represented as a *relational* attribute grammar (AG) [?]. The corresponding synthesized program is a *functional AG*. That is, the layout engine is a set of functions whose composition is syntax-directed by the document tree structure. We generalize modular synthesis to grammars of relations by handling alternative and recursive productions. To guarantee that our functional AGs are statically schedulable, we reject grammars with cyclic dependencies between attributes, thereby forbidding fixed-point computations.

Applications of GM Synthesis Even though GM synthesis was motivated by the challenges posed by layout, the techniques presented in this paper are generic: the algorithm is applicable to any hierarchical specification expressible as a relational AG. It could, perhaps, be used in modern hardware description languages (*e.g.*, Chisel [?]) where designs are trees of components with constraints on sizes, timings, etc.

We believe that small, domain-specific, *layout* languages (DSLL) can be both more expressive and more efficient than large—general-purpose—layout languages such as CSS. GM synthesis is part of the Programming by Manipulation (PBM) framework, a DSLL builder for non-programmers [?]. PBM infers layout specifications from user demonstrations, producing a relational AG which is then compiled to a tree-traversal engine using GM synthesis.

This paper makes the following contributions:

1. We present grammar-modular (GM) synthesis, a new technique enabling program synthesis to scale to previously intractable problems. We exploit the hierarchical structure of tree-shaped specifications to create a decomposition. We perform synthesis on each part individually, and combine the resulting functions into a program satisfying the overall specification.
2. We apply GM synthesis to generation of layout engines for DSLLs. We demonstrate empirically that our synthesized engines are sufficiently complete and are up to 200 times faster than Z3, a general-purpose constraint solver.
3. For constraints expressible as linear equations, we state a sufficient condition on the decomposition of the specification guaranteeing the completeness of GM synthesis.

2. Background and Motivation

In this section, we motivate our choice of non-directional constraints for layout specifications. We illustrate the flexibility and expressiveness of relational AGs compared to CSS [?], perhaps the most widely used layout language based on functional AGs. In the process, we introduce the key layout concepts and describe the inputs and outputs of our GM synthesizer.

The Sidebar Layout Problem Assume we want to lay out a document composed of two panels: a sidebar and a main area. Both contain malleable content (*e.g.*, text), whose aspect ratio can be altered. The corresponding document tree is made of a top-level horizontal divider with two children, one per panel. We want to make the sidebar wide enough to display all of its content on exactly one screen vertically without over/underflow. 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.

Layout Specifications To specify the layout semantics of our document, we assign *blocks* to document nodes. Blocks define a set of attributes (*e.g.*, positions and sizes) which decorate document nodes. Blocks also place either update functions (functional AG) or non-directional constraints (relational AG) on these attributes. Some attributes can be marked

```
Inputs=hbox.α, hbox.w

S ::= hbox(B, M) with
  B.w ← hbox.α*hbox.w
  M.w ← (1 - hbox.α)*
        hbox.w
B ::= div() with
  div.h ← f(div.w)  ★
M ::= <same as B>
```

(a) Overflowing f-AG (CSS)

```
S ::= hbox(B, M) with
  B.w + M.w == hbox.w

B ::= bar() with
  bar.h == f(bar.w)
M ::= main() with
  main.h == f(main.w)
```

(b) Sidebar DSLL r-AG

```
Inputs=hbox.h, hbox.w

S ::= hbox(B, M) with
  M.w ← hbox.w - B.w
  B.h ← hbox.h
B ::= bar() with
  bar.w ← f^-1(bar.h)
M ::= main() with
  main.h ← f(main.w)
```

(c) Synthesized sidebar f-AG

```
Inputs=bar.w, main.w

S ::= hbox(B, M) with
  hbox.w ← B.w + M.w
  hbox.h ← B.h
B ::= bar() with
  bar.h ← f(bar.w)
M ::= main() with
  main.h ← f(main.w)
```

(d) Synth. resizing sidebar f-AG

Figure 2. Four layout languages for the sidebar problem. Capital letters are non-terminals, *hbox*, *div*, *bar*, and *main* are blocks. Language (a) is the essence of CSS. It always computes height as a function of width, thus cannot express our sidebar design. Language (b) is non-directional which allows it to express our sidebar. From (b) and the input set $\{hbox.w, hbox.h\}$, we synthesize a functional AG (c) computing the sidebar layout, From (b), we can also produce an alternative functional AG recomputing the layout from a different set of inputs, for instance when the user resizes both panels (d).

as *input*; these are runtime values unknown at compile time, *e.g.* the size of an image, or the screen size. Solving the document layout amounts to computing the values of all attributes given input values, in accordance with the blocks' semantics.

Surprisingly, our simple example is impossible to implement with CSS. Figure 2a shows the simplified specification of the two CSS blocks relevant to our example. Since CSS is based on a functional AG, blocks semantics are expressed with functions. The sidebar must compute its width from its height. However, CSS always computes height as functions of width (★ in Figure 2a). The solving efficiency of CSS hinges on this restriction: it enables a tree-traversal engine where widths are computed in a traversal preceding all height computations. As such, even if CSS were extensible (it is not), one could not add a new sidebar block computing its width from its height.

By specifying the layout semantics with constraints instead of functions, we decouple the layout properties from their computation. The flow of computation can now be tailored to the particular type of layouts targeted by our DSLL. Figure 2b shows a relational AG defining two blocks (*bar*, *main*). A relationship between width and height is given, but which of these attributes is computed first is left open. Given a DSLL specified with a relational AG (Figure 2b) and a set of input attributes, our synthesizer outputs a functional AG (Figure 2c), capable of computing the layout of all derivable documents. Non-directional constraints gave us the flexibility to compute width from height for the sidebar and vice versa for the main panel.

In interactive layouts, the choice of input attributes depends on user actions. In the scroll box example, when the user moves the slide bar, its position becomes the input from which the layout engine recomputes the textbox position. From the same relational AG, we synthesize multiple layout engines, one per user interaction. Figure 2d shows another functional AG, also synthesized from Figure 2b, recomputing the layout when the user sets the width of both panels by dragging the middle separator. Relational AGs capture multiple layout scenarios in a single specification.

Paper Overview This paper is organized as follows: In Section 3, we introduce GM synthesis for single documents. Section 4 generalizes our algorithms to grammars of documents and discusses the completeness of our approach. Finally, we present our experimental results on synthesis of layout engines (Section 5) and discuss related work (Section 6).

3. Modular Synthesis

This section describes our modular synthesis in the single-document setting. The following section generalizes the algorithm to grammars.

3.1 Preliminaries

We start by defining functional synthesis [**?**], an instance of the AE-paradigm [**?**].

Functional Synthesis Let $R(i_1 \dots i_n, o_1 \dots o_m)$ be a relational constraint. Let $I = [i_1 \dots i_n]$ and $O = [o_1 \dots o_m]$ be dedicated *input* and *output* variables of R. Assume that R is functional in I. The functional synthesis problem is to find m total functions $f_1, \dots, f_m$ that compute the outputs from any valuations of inputs on which R holds: $R(I, O) \Rightarrow R(I, f_1(I), \dots, f_m(I))$. Because R is functional in I, all functions $f_1, \dots, f_m$ are semantically unique but may have multiple implementations, if one exists. We say that the set of functions $\{f_1, \dots, f_m\}$, denoted $f[I, O]$, *implements* R with respect to inputs I. In the context of layout, variables range over $\mathbb{R}$ and are called *attributes*.

We write $\pi_{I,O}(R)$ to denote a call to a procedure that returns a function that implements R; the procedure fails if the function does not exist. GM synthesis relies on a functional synthesizer (π) to perform synthesis on the subproblems created by decomposing the specification (Figure 3). π can be implemented using existing techniques (see Section 5).

The techniques presented in this paper are independent of the logical theories used to express constraints. Our examples and implementation rely on polynomial equations and linear inequalities over reals, augmented with basic trigonometric functions as well as min/max operators (see Section 5).

Empirically, we found such constraints expressive enough to specify a wide class of layouts and visualizations.

Layout semantics We define layout semantics of documents using attribute grammars (AGs), which formally capture attribute computation on a tree by attaching semantic rules to productions of the grammar. In functional AGs [**?**], these rules are functions. In relational AGs [**?**], the rules are arbitrary relations.

Definition 1 (Block)**.** *A* block *is a pair (V, R), where V is a finite set of attributes and R is a relational constraint over V. Some attributes of V are marked as* inputs, i.e. *their value will be provided externally when the layout is computed. We assume that R is in CNF, i.e., $R = cl_0 \wedge \ldots \wedge cl_n$. This CNF structure determines how R is decomposed into subproblems.*

Definition 2 (Document)**.** *A* document *is a tree of block-labeled nodes derived from a relational AG.*

Definition 3 (Language)**.** *A* language of documents *$\mathcal{L}$ is the set of documents generated by a relational AG $\mathcal{G}$.*

Since documents are trees, technically $\mathcal{G}$ is a regular tree grammar [**?**] instead of a word grammar[1]. The significant change is that productions of $\mathcal{G}$ have the form

```
A ::= hbox(B,C)
```

where *hbox* is a terminal symbol labeling the tree node. In our relational AGs, terminals refer to blocks of constraints.

We distinguish a special kind of semantic rules: those placing constraints between attributes from a parent and its children. Such rules are called *connections*. For instance, the first rule is a connection, while the second one is not.

```
S ::= a(B)  with a.x = B.x
B ::= b()   with b.x = f(b.y, b.z)
```

This distinction will become important when we decompose documents at node boundaries. Without loss of generality, we assume that a connection c, denoted by $(A, B)_c$ is an equality constraint between two sets of attributes A and B.

The layout of a document d from a relational AG $\mathcal{G}$ is the solution to $rel(d)$, the constraint system generated by $\mathcal{G}$.

Finally, given a document d, let I_d be the set of input attributes of d. Let O_d be all other (non-input) attributes of d. We are now ready to define document engines.

Definition 4 (d-Engine)**.** *Given a document d, a d-*engine *is a function $f[I_d, O_d]$ which implements $rel(d)$.*

Synthesis of Layout Engines To synthesize a d-engine for a particular document, the simplest approach would be directly computing $\pi_{I_d,O_d}(rel(d))$. This is impractical for all but the most trivial documents (see Section 5).

Given a document d, we synthesize a d-engine in three steps (Figure 3): (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-engine. Before we detail each of the three steps, we highlight the algorithmic challenges by constructing a d-engine for a small document.

[1] Regular tree grammars can be thought as generating the set of derivation trees of a context-free word-grammar [**?**].

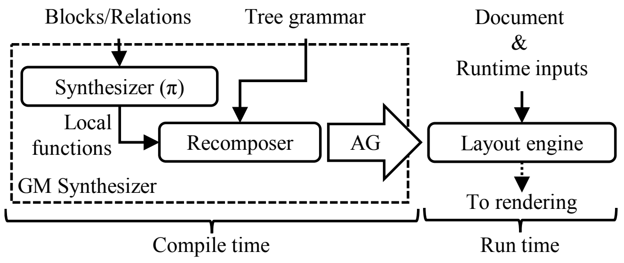


Figure 3. GM synthesizer. The first step of GM synthesis—decomposition—is not shown. The AG scheduler is out of the scope of this paper. Its output is the layout engine itself.

3.2 Example of d-Engine Synthesis

Let us consider a document comprised of 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:

$$V_a \stackrel{\text{def}}{=} \{x, y, z, i\} \qquad R_a \stackrel{\text{def}}{=} x = i \wedge i + z = y$$
$$V_b \stackrel{\text{def}}{=} \{x, y\} \qquad R_b \stackrel{\text{def}}{=} x^2 = y$$

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

Decomposition (Step 1) To decompose $rel(d)$, we follow the document structure and create two subproblems, R_a and R_b, one per node of the document.

Local Synthesis (Step 2) We generate local functions for each node of the document. To do so, we need to (i) partition each block relation into subsets of clauses; and (ii) partition attributes of each block into input/output sets. For the sake of the example, we use an oracle to coordinate these two local decisions. Then, we synthesize local functions for each of set of clauses using our functional synthesis procedure π.

For our example document, block a is made of two clauses: $x = i$ and $i + z = y$. The oracle partitions R_a into subsets $s_0 \stackrel{\text{def}}{=} \{x = i\}$ and $s_1 \stackrel{\text{def}}{=} \{i + z = y\}$. Then the oracle partitions V_a into an input set $I_a \stackrel{\text{def}}{=} \{i, y\}$ and an output set $O_a \stackrel{\text{def}}{=} \{x, z\}$. Given these two partitions, we attempt to generate local functions for each set of clauses s_0 and s_1 using π. In this case, $\pi_{I_a,O_a}(s_0)$ yields the function $f_1 \stackrel{\text{def}}{=} x := i$, and $\pi_{I_a,O_a}(s_1)$ produces $f_2 \stackrel{\text{def}}{=} z := y - i$

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 $f_3 \overset{\text{def}}{=} y := x^2$.

We now have a set of local functions for each node of the document

Recomposition (Step 3) The third step consists of constructing a global function implementing $rel(d)$ 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 available local functions (Figure 4). 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.

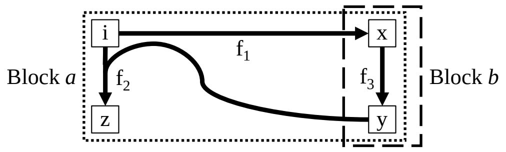


Figure 4. The hypergraph of the dependencies of f_1, f_2, and f_3. Note that the local function f_2 is represented by a hyperedge with two sources: i and y.

Implementing the Oracle Let's take a step back to analyze the role of the oracle. We relied on it 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 examining 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. Intuitively, each subset of clauses corresponds to one "pass" of the global function through the corresponding block.

We used the oracle to simplify our synthesis algorithm which *conceptually* relies on global reasoning to synthesize local functions. We actually synthesize local functions considering both all partitions of clauses into subsets and all partitions of attributes into input/output sets. We then "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.

3.3 The Algorithm

We formalize the three steps of GM-synthesis (decomposition, local synthesis, and recomposition) for a document d.

Decomposition (Step 1) 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 the completeness of GM synthesis in Section 4.1.

Local Synthesis (Step 2) We start by defining local functions formally.

Definition 5 (Local Function)**.** *Given a block* $(V, cl_0 \wedge \ldots \wedge cl_n)$, *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 implements 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 all possible local functions, for each block (V, R) in d, we enumerate all partitions of clauses of R as well as all input/output partitions of V, as detailed in Algorithm 1.

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 6 (Hypergraph Summary)**.** *Given a document d, an hypergraph summary* $H_d \overset{\text{def}}{=} (V, E)$ *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*

Algorithm 1: Synthesize local functions for a block.

```
Input: A block $b \overset{\text{def}}{=} (V, cl_0 \wedge \ldots \wedge cl_n)$
Output: A set of local functions over attributes $V$
$L \leftarrow \varnothing$
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 $L$.
return $L$
```

(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 represent them with local functions. For a single document, each connection (A, B) is such that A and B are singletons. 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 2 details this process.

Algorithm 2: Construct 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 $d$
$E \leftarrow \varnothing$
foreach node $n$ in $d$ labeled with block $b$ do
    Add $\{(I, O) \mid (f, I, O, S) \in \text{Algo1}(b)\}$ to $E$.
foreach connection $(A, B)$ in $C$ do
    Add $\{(A, B), (B, A)\}$ to $E$.
return $(I_d \cup O_d, E)$
```

Before we define the d-engine 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 1. *Each acyclic 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 $O_p = \bigcup_{0 \le i \le n} O_i$ and $I_p = (\bigcup_{0 \le i \le n} I_i) \setminus O_p$.*

From hyperpath properties [?], 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 $I_i \subseteq \bigcup_{0 \le j \le i-1} O_j \cup I_p$.
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 2. *Let $p = f_0 \ldots f_n$ be an acyclic hyperpath in H_d representing f_p. Let (f_i, I_i, O_i, S_i) be the ith function in p. Then f_p implements $\bigwedge_{0 \le i \le n} S_i$. We say that f_p satisfies all clauses* traversed.

Lemma 2 follows directly from the fact that, by construction, each local function (f_i, I_i, O_i, S_i) implements S_i. Finally, let us define the subset of paths which can be executed.

Definition 7 (Executable Path)**.** *A hyperpath p in H_d is* executable *iff it starts from the document inputs which is when 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-engine. That is, a global function which implements $rel(d)$ with respect to the document inputs I_d.

Definition 8. *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 tree p in H_d encodes a global function which implements $rel(d)$ 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 2, we conclude that f_p computes all attributes of 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 5).

The converse is not true; GM synthesis assumes that the global function is expressible as compositions of local functions.

Together, Theorems 1 and 2 show that our approach is correct: the d-engines synthesized always fulfil the specification. Note that finding a spanning tree in a hypergraph is NP-complete [?]. In the next section, we explain how to encode the search for a d-engine in SMT after generalizing our approach to languages of documents.

4. Grammar-Modular Synthesis

We generalize the modular synthesis technique presented so far to grammar-modular synthesis.

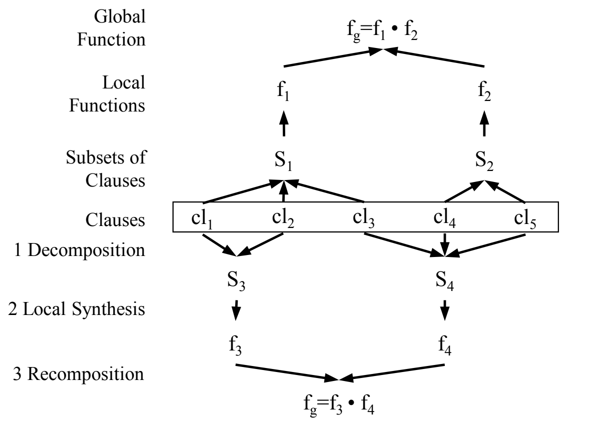


Figure 5. The three steps of GM synthesis. This diagram shows that two distinct decompositions can lead to syntactically different, yet semantically equivalent, d-engines.

To support grammars producing more than one document, we need to handle recursive productions and non-terminals with more than one production. We start by formally defining language engines.

Definition 9 ($\mathcal{L}$-Engine). *Let $\mathcal{L}$ be the language of documents induced by the relational AG $\mathcal{G}$. An $\mathcal{L}$-engine is a statically schedulable functional AG that defines a d-engine for every document $d \in \mathcal{L}$.*

In functional AGs, the *mode* of an attribute is either inherited or synthesized [?]. To construct an $\mathcal{L}$-engine from $\mathcal{G}$, we compute: (i) the mode of all attributes together with a corresponding subset of local functions; and (ii) a total order over attributes. The total order prevents cyclic dependencies, which guarantees that the resulting functional AG is statically schedulable.

Synthesizing $\mathcal{L}$-Engines We synthesize an $\mathcal{L}$-engine in three steps: First, we create a *witness* document d_w which exhibits all productions of $\mathcal{G}$. The hypergraph summary (Algorithm 2) of d_w is a witness of all documents in $\mathcal{L}$. 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 an $\mathcal{L}$-engine.

Let d_w be the witness document. Since $\mathcal{G}$ is based on a regular tree-grammar, d_w can be easily produced by systematically unrolling $\mathcal{G}$ until every production is taken. By doing so, we ensure that $rel(d_w)$ contains all constraints of $\mathcal{G}$.

For non-terminals with multiple productions, we must ensure that, for all productions, values flow in the same direction (either up or down) through each connection. Conveniently, we can exploit properties of hyperpaths to this end. For example, consider the following language where block a may have either block b_1 or b_2 as child.

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

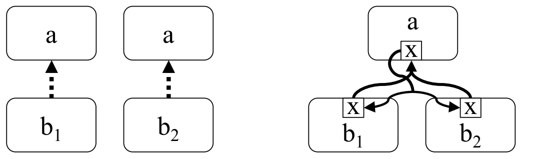


Figure 6. A language of two documents, each stemming from a two-production grammar (a), Algorithm 2 encodes the connection $(\{a.x\}, \{b_1.x, b_2.x\})$ with two hyperedges, thereby enforcing the same flow of computation for both documents (b).

Attribute $a.x$ is connected to either $b_1.x$ or $b_2.x$. We encode the two 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 2 encodes c with two hyper-edges (one with two destinations and one with two sources) forcing values to flow either up or down through both derivations (Figure 6). We encode connections to multi-production non-terminals directly with hyperedges with multiple sources and destinations.

To handle recursive productions, we (1) introduce cycles into the witness document; and (2) relax the definition of covering spanning trees (Definition 8) in order to distinguish between cycles due to recursion in the grammar and dependence cycles among attributes that would exist in a single document.

SMT Encoding We encode the existence of an $\mathcal{L}$-engine as an SMT query using boolean and integer arithmetic.

Let $H_{d_w} = (V, E)$ be the hypergraph summary of d_w. 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. 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 $\mathcal{L}$-engine; we say that f is *selected*. To model inputs, we augment F with one function $(-, \varnothing, I_{d_w}, \varnothing)$ attached to the flag e_{inputs}.

We encode each attribute $x \in V$ with two variables: (i) one boolean m_x representing the mode of x, either inherited ($\downarrow$) or synthesized ($\uparrow$); and (ii) one integer l_x used to impose a total order on all attributes.

We partition the connections of $\mathcal{G}$ 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 $\mathcal{G}$, we encode each clause cl of R with one boolean named e_{cl}.

Finally, we define $bm(x)$, 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 analogous to modes of logic programs: attributes marked *in* are computed outside the block and propagated to it by connections; attributes marked *out* are computed within the block by a local function.

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

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.

$$\phi_{\text{Conn}}((A,B)_c) := \big(m_c = \downarrow \Rightarrow \textstyle\bigwedge_{x \in B} m_x = \downarrow\big) \quad \wedge \\ \big(m_c = \uparrow \Rightarrow \textstyle\bigwedge_{x \in A} m_x = \uparrow\big)$$

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 $\mathcal{L}$-engine 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.

$$\phi_{\text{Fun}}(f,I,O,S) := e_f \Rightarrow \bigwedge_{x \in O} bm(x) = out \ \wedge \ \bigwedge_{cl \in S} e_{cl}$$

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 $\mathcal{L}$-engine. 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 an $\mathcal{L}$-engine 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. 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. We must distinguish cycles in d_w representing recursive computation of attributes—which unwind on a document, thus are safe—from those denoting true cyclic dependencies. Consider the following grammar which computes the height of a tree with blocks $a \overset{\text{def}}{=} (\{x\}, true)$ and $b \overset{\text{def}}{=} (\{x\}, true)$:

```
S ::= a(S) with a.x == S.x + 1
   |  b()  with b.x == 1
```

This language admits a sound $\mathcal{L}$-engine. Its witness document contains a cycle: from $a.x$ to itself. This cycle unwinds on all documents because it contains one recursive connection. That is, there are no cyclic dependencies if every cyclic path in the subgraph of selected local functions includes one recursive connection.

$$\phi_{\text{Sched}} := \bigwedge_{(A,B)_c \in N} \big(m_c = \downarrow \Rightarrow \textstyle\bigwedge_{x \in B} l_x > \max_{y \in A}(l_y)\big) \ \wedge \\ \bigwedge_{(A,B)_c \in N} \big(m_c = \uparrow \Rightarrow \textstyle\bigwedge_{x \in A} l_x > \max_{y \in B}(l_y)\big) \ \wedge \\ \bigwedge_{(f,I,O,S) \in F} \big(I \supset \varnothing \wedge e_f \Rightarrow \textstyle\bigwedge_{x \in O} l_x > \max_{y \in I}(l_y)\big)$$

Soundness To guarantee that the $\mathcal{L}$-engine implements $rel(d_w)$, the local functions selected must: (i) traverse all the clauses of (all the blocks of) $rel(d_w)$; and (ii) contain the function modeling inputs.

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

$\mathcal{L}$-Engine 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) as well as modes for all attributes (m_x) and all connections (m_c).

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

Theorem 3 (Correctness)**.** *All models of ϕ are $\mathcal{L}$-engines.*

The translation of models of ϕ to functional AGs is straightforward: The e_f booleans indicate which local functions to use.

4.1 Completeness

GM synthesis is incomplete, thus might fail to find an engine, even when one exists. In Section 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. 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.

We state a condition for hierarchical linear systems of equations sufficient to guarantee zero cost of modularity. For clarity, we consider a single document; the condition is generalizable by induction on the language grammar. 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 4 (Completeness Condition)**.** *For linear equations, GM synthesis has no cost of modularity if M_d can be triangularized using row combinations (i.e., adding a linear combination of rows to another) only between rows belonging to the same block, and row interchanges for any pair of rows.*

5. Evaluation

We evaluate GM synthesis along three axes:

- *Scalability vs. Completeness* GM synthesis trades completeness for scalability: is it both scalable and sufficiently complete to synthesize $\mathcal{L}$-engines for realistic DSLLs?
- *Performance* How does the solving speed of the synthesized $\mathcal{L}$-engines compare with the speed of state-of-the-art, general-purpose constraint solvers?
- *Parameterizable* DSLL*s* Can our layout specifications produce $\mathcal{L}$-engines for a diverse range of user interactions (*e.g.*, screen resizing; data updates), each updating different input variables?

Experimental Setup GM synthesis is parametrized by the local synthesis procedure π. In our experiments, we implemented π with a combination of well-known techniques: For linear relations, we used a CEGIS loop [**??**] iteratively trying templates of the form *if* $(l_1 > 0)$ l_2 *else* l_3 where l_1, l_2, and l_3 are linear combinations of attributes. Synthesis and verification queries are encoded in SMT linear real arithmetic [**?**]; attribute values are not bounded. Polynomial equations are solved in isolation, using a set of algebraic rewrite rules. Other tools could be used to implement π, for example Comfusy [**?**] and Sketch [**?**].

We used the Superconductor AG scheduler [**?**] to compile $\mathcal{L}$-engines to (sequential) tree traversals. The resulting traversals are implemented in JavaScript and operate directly on the browser DOM. As a result, our custom $\mathcal{L}$-engines can easily be deployed in any web browser. Figures 1b and 1c have been laid out by one of our $\mathcal{L}$-engines. All our benchmarks were run on a 2.5GHz Intel Sandy Bridge processor with 8GB of RAM using Firefox 30.0.

Case Studies To show that GM synthesis is widely applicable, we evaluate it on three DSLLs constructed with PBM [**?**], one for each of the three major layout domains: (i) document (webpage) layout; (ii) GUI; and (iii) data visualization. Each language is full-fledged in that it computes all attributes required by rendering.

1. Our first case study is a guillotine language where a set of horizontal and vertical dividers partition the space. Such a language can encode a subset of CSS [**?**]. The guillotine language totals 30 linear constraints, and satisfies the completeness condition (Theorem 4).
2. Our second case study is a language of flexible grids [**?**]. Such languages are frequently used to layout widgets in graphical user interfaces [**?**]. The sizes of each cell of the grid are allocated based on a weighted sum. The weight of each cell is a runtime input, which produces non-linear constraints. The grid language consists of 47 constraints.
3. Finally, a language of treemaps [**?**], 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). Each leaf has a runtime input corresponding to its relative area. Constraints involving area computations are non-linear. The treemap language has 40 constraints.

To illustrate our DSLL specification language, we show below a partial definition of the treemap block that tiles the space horizontally. Lines 2 and 3 set up the relative coordinates *left*, *right*, denoting the relative horizontal displacement from the parent node, based on the absolute coordinate *x*. The third constraint is key: it binds the visual area of each document node to the value of the tile. These constraints are local in that they refer to parent and children in the document.

```
1 block hdiv(...) {
2     x == parent.x + left
3     left + width == right
4     scale * value == height * width
5     child1.left >= child0.width...
6 }
```

Scalability and Completeness Our GM synthesizer is sufficiently complete to successfully generate an $\mathcal{L}$-engine for each of the three case studies. Synthesis took less than four minutes in each case, an acceptable compilation time, with the local synthesis and the recomposition steps taking approximately equal time. Table 1 illustrates the complexity of the $\mathcal{L}$-engines obtained after scheduling. Listed are the number of tree traversals, the number of local functions used, as well as the size of the JavaScript code. For comparison, Mozilla's new Servo browser employs four passes to lay out CSS [**?**]. The number of local functions is per grammar ($\mathcal{L}$-engine), rather than for a document. Finally, the number of lines of code reported includes only the layout engine itself (*i.e.*, the computation of document attributes); the rendering code has been excluded.

GM synthesis provides no guarantee that the $\mathcal{L}$-engines are optimal, neither in the number of traversals nor in the number of operations. We manually checked each $\mathcal{L}$-engine: all are optimal in the number of traversals.

We also compare GM synthesis with non-modular functional synthesis methods, specifically with Comfusy and Sketch. These techniques are limited to synthesis of d-engines

Language	Traversals	Local Functions		SLOC	Time
		Total	Selected		
Guillotine	t	289	74	317	126
Grid	t ; b ; t	385	89	483	175
Treemap	t ; b ; t ; b ; t	394	91	599	194

Table 1. The complexity of our $\mathcal{L}$-engines. The second column shows the number and type of tree passes: t and b denote top-down and bottom-up passes, respectively. The third column reports the number of local functions synthesized and the subset used. The last two columns show the number of lines of code and the total synthesis time in second.

(*i.e.*, they do not generalize to languages of documents), hence we asked them to synthesize a solver for a single document of 127 nodes. Both systems failed to synthesize a d-engine in less than one hour. These results suggest that GM synthesis may strikes a good balance between completeness and scalability in the domain of layout engines.

Performance We compare the performance of our $\mathcal{L}$-engines with Z3 [?], a state-of-the-art constraint solver. Our engines are implemented in JavaScript, a relatively slow language. Z3 solves the constraint system defined by the document ($rel(d)$) at runtime. We tested several solver algorithms implemented in Z3.

We measured the time to compute the layout of documents from 255 to 16383 nodes, for each of our DSLLs. Such document sizes are typical [?]. For reference, the front page of `nytimes.com` contains over 3000 nodes and data visualizations tend to be larger. Our benchmark documents are balanced trees generated randomly. For Z3, we chose the fastest SMT theory which could express the layout specification. Interestingly, the non-linear real arithmetic solver was faster than 16bit bitvectors for both the grid and treemap languages. For guillotine, we used linear real arithmetic. Table 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 2. Time to compute the layout in millisecond. Missing entries (—) indicate "unknown" answers (no model). Notice that our $\mathcal{L}$-engines scale linearly with the document size.

Our $\mathcal{L}$-engines scale linearly with size of the document, whereas Z3 fails on the largest document (either timing out or reporting "unknown") for all three case studies. This 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), $\mathcal{L}$-engines are between 39 and 231 times faster than Z3. Our results show that across the three case studies, our $\mathcal{L}$-engines are fast enough (<0.5 second) for interactive settings.

***Parameterizable* DSLLs** We illustrate the expressiveness of non-directional constraints by synthesizing multiple $\mathcal{L}$-engines from the same DSLL, each parameterized by a different set of runtime inputs. Each engine recomputes the layout in response to some event triggered by a user interaction. Each event sets the values of some runtime inputs, from which all remaining attributes are computed.

We illustrate the expressive power of non-directional constraints on the language of treemaps. Assume that the treemap represents the market capitalization of some companies. The leaves of the document are companies while inner nodes encode the tiling of the screen (Figure 1a). We consider three events: (i) the values of all companies are updated; (ii) the user resizes the treemap; and (iii) the user moves the treemap.

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 our treemap DSLL, our synthesizer generates three $\mathcal{L}$-engines, one per set of runtime inputs (Table 3). Note that the engines are dramatically different from each other in that they require a different number of tree passes.

Event	Inputs	Traversals	SLOC
New market cap.	$t.value$ $\forall$ tiles t	5	599
Resize root	*root.w*, *root.h*	3	463
Reposition root	*root.x*, *root.y*	1	165

Table 3. Three $\mathcal{L}$-engines, one per event, recomputing the treemap layout from different sets of inputs. For space reasons, we only show input attributes with new values and omit those which are input but remain constant. We report the number of tree-traversals and the size of code.

To conclude, we have shown empirically that (i) GM synthesis is both scalable and complete enough to generate $\mathcal{L}$-engines for a variety of DSLLs; (ii) our $\mathcal{L}$-engines outperform general-purpose constraint solvers; and (iii) relational AGs are a concise formalism for expressing interactive layouts with multiple flows of computation.

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 [**??**], is an instance of the AE-paradigm, also known as the Skolem paradigm for synthesis [**?**]. GM synthesis builds upon functional synthesis procedures, such as Comfusy [**?**] or Sketch [**?**], by enabling modular decompositions of specifications to gain scalability. **?** also propose a modular synthesis technique but use a different form of modularity.

Constraint Planning (CP) The task of finding a d-engine can be cast as a multi-way (*i.e.* non-directional) constraint planning problem for which solvers like SkyBlue [**?**] and QuickPlan [**?**] have been proposed. In CP, each "planning constraint" corresponds to a set of clauses in our framework. Similar to our d-engine 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 3.2, 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-engines, 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 [**?**].

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

Attribute Grammar Our modular synthesis algorithm has close connections with relational AGs and logic programming. **?** give theoretic constructions demonstrating how relational grammars, functional grammars and directed clause programs are related to one another. Mode analysis [**?**] techniques for logic programs, which compute whether clause arguments of logical programs are input or output, could be—in principle—transposed to AGs 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 $\mathcal{L}$-engines, 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 DSLLs. Mode analysis techniques based on abstract interpretation operate on the whole program.

Constraint Logic Programming (CLP) In CLP [**???**], constraint systems are flat and unstructured while we exploit the tree structure to produce $\mathcal{L}$-engines 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).

7. Conclusion

We presented grammar-modular synthesis, a new algorithm exploiting the structure of hierarchical specifications to scale synthesis to large relations at the cost of completeness. We synthesized tailored layout engines for custom languages of documents. 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 $\mathcal{L}$-engines generated outperform general-purpose constraint solvers by one order of magnitude. 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 $\mathcal{L}$-engines.

We are interested in applying GM synthesis to domains beyond document layout. For instance, the techniques presented in this paper could potentially generate parametrized hardware designs from trees of components with constraints such as size or delay.

References

Alur, R., Bodík, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE.

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.

Ausiello, G. (1988). Directed hypergraphs: Data structures and applications. In Dauchet, M. and Nivat, M., editors, *CAAP '88*, volume 299 of *Lecture Notes in Computer Science*, pages 295–303. Springer Berlin Heidelberg.

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. (2012). Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA. ACM.

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

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

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`.

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.

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.

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

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, pages 231–241, 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.

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.

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.

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.

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.

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.

Singh, R., Singh, R., Xu, Z., Krosnick, R., and Solar-Lezama, A. (2014). Modular synthesis of sketches using models. In *Verification, Model Checking, and Abstract Interpretation*, volume 8318 of *Lecture Notes in Computer Science*, pages 395–414. Springer Berlin Heidelberg.

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.

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

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