Conflict-Driven Synthesis for Layout Engines

JUNRUI LIU, University of California, Santa Barbara, USA
YANJU CHEN, University of California, Santa Barbara, USA
ERIC ATKINSON, Massachusetts Institute of Technology, USA
YU FENG, University of California, Santa Barbara, USA
RASTISLAV BODIK, Google Research, USA

Modern web browsers rely on layout engines to convert HTML documents to layout trees that specify color, size, and position. However, existing layout engines are notoriously difficult to maintain because of the complexity of web standards. This is especially true for incremental layout engines, which are designed to improve performance by updating only the parts of the layout tree that need to be changed.

In this paper, we propose Medea, a new framework for automatically generating incremental layout engines. Medea separates the specification of the layout engine from its incremental implementation, and guarantees correctness through layout engine synthesis. The synthesis is driven by a new iterative algorithm based on detecting conflicts that prevent optimality of the incremental algorithm.

We evaluated Medea on a fragment of HTML layout that includes challenging features such as margin collapse, floating layout, and absolute positioning. Medea successfully synthesized an incremental layout engine for this fragment. The synthesized layout engine is both correct and efficient. In particular, we demonstrated that it avoids real-world bugs that have been reported in the layout engines of Chrome, Firefox, and Safari. The incremental layout engine synthesized by Medea is up to 1.82× faster than a naive incremental baseline. We also demonstrated that our conflict-driven algorithm produces engines that are 2.74× faster than a baseline without conflict analysis.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: program synthesis

ACM Reference Format:
Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodik. 2023. Conflict-Driven Synthesis for Layout Engines. *Proc. ACM Program. Lang.* 7, PLDI, Article 132 (June 2023), 22 pages. https://doi.org/10.1145/3591246

1 INTRODUCTION

Layout engines compute the visual appearance of web pages. They are at the heart of web browsers and numerous GUI frameworks ([Github 2013]). The ubiquity of layout engines makes layout bugs especially problematic. These bugs can make applications confusing to navigate, or simply unusable. Unfortunately, layout engines contain a large number of unresolved layout bugs, some of which have remained open for over five years [Brubeck 2014; Keesara 2007].

Those bugs may resist simple fixes due to deep-rooted mismatches between intended semantics of a layout feature and the engine's architecture. For example, the bug [stshine 2016] from the Servo layout engine is caused by incorrect interaction of two features: margin collapse and text

Authors' addresses: Junrui Liu, junrui@cs.ucsb.edu, University of California, Santa Barbara, USA; Yanju Chen, yanju@cs.ucsb.edu, University of California, Santa Barbara, USA; Eric Atkinson, eatkinson@csail.mit.edu, Massachusetts Institute of Technology, USA; Yu Feng, yufeng@cs.ucsb.edu, University of California, Santa Barbara, USA; Rastislav Bodik, rastislavb@google.com, Google Research, USA.

2475-1421/2023/6-ART132
https://doi.org/10.1145/3591246

layout. These features must pass information during layout computation but they are implemented in Servo as modules that run independently. The unresolved circular dependency is the root cause of the bug.

Such layout bugs cannot be fixed by permuting a few lines of code, because a correct implementation must divide the code for margin collapsing and text layout into multiple interleaving steps. This would alter the existing *layout schedule*, which is the order in which the visual attributes of a layout are computed. However, layout schedules are the largest-scale architectural decision in a browser layout engine: once chosen, the schedule is difficult to change without a complete rewrite. For the aforementioned bug, changing the layout schedule would distribute the margin collapsing code across multiple phases, affecting a large number of files and data structures. A Servo developer remarked [stshine 2016] that "it took three weeks even before I realize[d] the actual complexity of the problem." In the end, the Servo developers decided to delay fixing this bug until a complete rewrite of the layout engine is done.

Besides the difficulty of changing the layout schedule, designing a correct layout schedule from the ground up is also difficult. The semantics of web layout contain complex dependencies and interacting features, precluding a manual analysis. Thus, even if the layout schedule could be changed to resolve a specific bug, there is little guarantee that the change would not inadvertently introduce new bugs.

Beyond correctness, ensuring the performance of layout engines is equally important. Specifically, *incrementality* is crucial to the performance of real-world layout engines: once the attributes on a layout tree have been computed, small changes should not trigger a full re-layout; instead, an incremental layout engine should, ideally, only recompute the attributes that are transitively affected by said changes. Engineering correct, efficient incremental traversals are especially challenging because developers need to reason about intricate conditions under which recomputation of certain elements can be safely skipped, while navigating the large space of valid incremental schedules to find an "optimal" one.

We propose to replace this painstaking development process with Medea, a tool for synthesizing layout engines with correctness guarantees while achieving high-performance via incrementality. While existing tools like Hecate [Chen et al. 2022] can technically synthesize correct layout engines by reducing the synthesis problem to *satisfiability queries* [Torlak and Bodik 2014], they can not deal with *optimization queries* required for synthesizing complex, incremental layout engines with competitive performance.

To use Medea, the developer specifies the layout semantics using an attribute grammar, and provides a sketch of the layout schedule containing holes yet to be filled with attribute computation rules drawn from the grammar. Both the attribute grammar and the layout schedule are highly customizable. Medea then proposes incremental layout schedules that are correct (i.e., they respect all attribute dependencies) and efficient (i.e., they reduce unnecessary recomputation in re-layouts) through its *optimal synthesis algorithm*. To reduce the synthesis time, Medea proposes a novel algorithm inspired by the successful conflict-driven learning approach [Feng et al. 2018; Silva and Sakallah 1997] used by automated theorem provers. Specifically, our algorithm analyzes conflicts that explain the inefficiency in a schedule and learns useful lemmas to prevent the synthesizer to repeat the same mistake.

We evaluate Medea on layout specifications derived from the Cassius [Panchekha and Torlak 2016] formalization of web layout. Medea successfully synthesizes layout schedules for this specification, and the resulting layout engines are free from dozens of real-world bugs collected from the layout engines of Chrome, Firefox, and Safari. Medea is also expressive in that it can handle complex layout features such as margin collapse, floating layout, and absolute position. The incremental layout engine synthesized by Medea achieves up to 1.82× speed-up compared to a

naive incremental baseline. Finally, we also demonstrate the effectiveness of our conflict-driven algorithm in that it leads to 2.74× faster schedules compared to a baseline without conflict analysis.

To summarize, we make the following contributions:

- We show how incremental layout algorithms can be expressed as sketches for non-trivial CSS features, i.e., how a space of incremental algorithms can be defined syntactically.
- We propose an efficient synthesis algorithm based on *conflict-driven learning*.
- We implement a tool called Medea that can synthesize modern layout engines free from dozens of real-world bugs collected from Chrome, Firefox, and Safari.

2 OVERVIEW

In this section, we illustrate how Medea works using a running example, adapted from a real-world layout bug [Walton 2014]. Before doing so, we first give a brief background on browser layout engines.

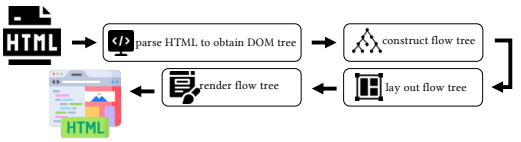

Fig. 1. Basic workflow of a layout engine

Background on browser layout engines. A browser layout engine transforms a web page from source code to pixels through successive tree transformations, as illustrated in Figure. 1. First, the engine transforms a HTML web page into a document object model (DOM) tree that encodes the structure of the objects contained in the web page. Then, the DOM tree is compiled into a *flow tree*, which abstracts and organizes HTML objects using the box model [W3C 2007]. Afterwards, styles are applied to the flow tree to determine, e.g., geometric attributes such as the dimension, location, and color of each node. Finally, the layout engine renders the page into pixels.

Our motivating example models *render trees*, which contains nodes with visual elements and computed structural and styling attributes. A node on a render tree may lay out itself and its children in various ways, e.g., vertically or horizontally. In this example, we consider a simplified version of the HTML *block-level elements*, which are stacked vertically (Figure. 2). When constructing the render tree, a layout engine computes various visual attributes of each node, such as the width, height, x- and y- coordinates, which may have dependencies among themselves and with other attributes. For example, the layout engine may compute those attributes as follows[1]:

- The width of a node is the width of the parent minus the node's left and right margins.
- The height of the parent is the accumulated height of its children.
- The x-coordinate of a node is the x-coordinate of its parent offset by the node's left margin.
- The y-coordinate of a node is the y-coordinate of its parent offset by the accumulated height of its preceding siblings and the node's top margin. However, real-world layout specification[2] introduces the following twist: if the node is the first among its siblings , then in some cases its top margin collapses into its parent's top margin. This CSS feature is known as *margin collapsing*, which is where the aforementioned layout implementation [Walton 2014] gets wrong[3].

Note that the above computation rules impose constraints on the order in which the attributes are computed. For example, the parent's width must be computed before the computation of each of the child's width, while the height of the parent depends on the accumulated height of the children.

[1]Those computation rules are standardized by the W3C CSS specification [W3C 2007]. Our motivating example slightly differ from the standards to simplify the presentation.

[2]https://www.w3.org/TR/CSS/

[3]The layout engine is required to determine whether the current node is a *block-formatting context*. The root cause of the incorrect implementation is failing to link the `overflow: scroll` CSS feature with the block-formatting context, thus resulting in the violation of the specification.

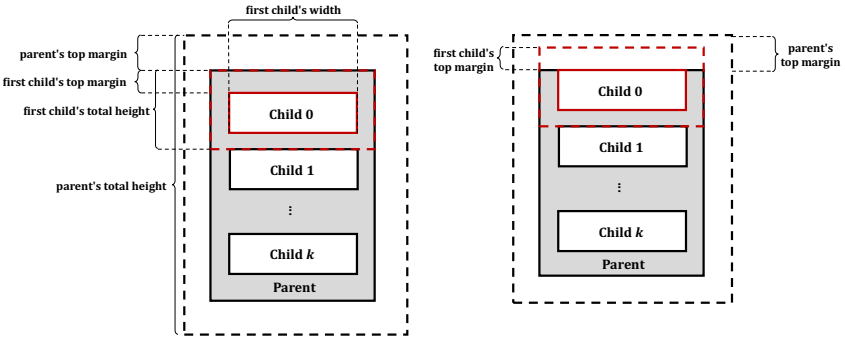

Fig. 2. A block element lays out its children vertically (left). Each element contains margins that separate it from the other elements. However, a feature called *margin collapse* allows the margins of two elements to overlap. On the right, the top margin of the parent overlaps with the top margin of its first child.

The layout engine must schedule the attribute computation in a way that respects such constraints. A correct schedule, e.g., can first perform a pre-order traversal over the render tree starting from the top level block to compute the width, followed by a post-order phase to compute the height. A layout engine that violates any of those constraints will attempt to read from attributes whose values are not yet computed. In a realistic render tree, however, manually searching for correct traversals is challenging, as each tree node may contain hundreds of attributes with intricate dependencies between them.

Besides correctness, modern layout engines achieve good *performance* using incremental layout. The idea of incremental layout is that, although the initial layout needs to compute every attribute, when a subset of the attributes are modified (due to, for example, external events such as mouse clicks), a rerun of the traversal only needs to recompute those attributes that transitively depend on the modified subset. Layout engines commonly implement this idea by introducing an additional set of attributes, called *dirty bits*, to keep track of which attributes hold values that might be different from the previous run. The code for an incremental traversal can be thought of as decorating the non-incremental traversal with conditionals, which are toggled on or off based on whether certain attributes have been marked dirty.

However, "incrementalizing" a layout engine requires careful thought, as they may introduce both correctness issues and performance overhead. For example, a naive incremental layout engine can associate every attribute with a unique dirty bit, and guard every attribute computation with the necessary dirty bits. This kind of incremental traversal achieves optimality in the sense that no redundant re-computation will be performed. However, it also comes with a prohibitive cost of maintaining a large number of dirty bits, as the number of attributes on each tree node can be on the order of hundreds or thousands.

Fortunately, many attributes are semantically related, allowing us to map multiple attributes to the same dirty bit, and to compute or skip groups of related attributes during an incremental layout. A smart design of both mappings from dirty bits to attributes and grouping of attribute computation as conditional blocks can offer substantial speedup over a "naive" incremental traversal, and is what makes incremental traversals efficient in practice. Nevertheless, achieving the best combination by manually exploring different schedules, while ensuring their correctness at the same time, is impractical and error-prone.

Layout engine synthesis with Medea. We have developed Medea, a tool that can automatically synthesize correct and efficient incremental layout engines. With Medea, the user encodes the layout semantics declaratively, in the form of attribute grammar [Knuth 1968], together with a

```
interface Viewport {
  input w, h : Pixels;
  output total.h: Pixels;
}

class CViewport : Viewport {
  children { r : LayoutBox; }
  rules {
    r.p.{x, y, w, h} = {0.0, 0.0, w, 0.0};
    total.h := r.total.h;
  }
}

interface LayoutBox {
  input is_first: Bool;
  input is_bfc: Bool;
  input margin.{left, right, top, bottom} : Pixels;
  output uncollapsed_top : Pixels;
  output should_collapse: Bool;
  output p.{x, y, w, h, top, is_bfc} : Pixels;
  output box.{x, y, w} : Pixels;
  output total.h: Pixels;
  output h_acc : Pixels;
}

class Block : LayoutBox {
  children { cs : [LayoutBox]; }
  rules {
    box.x := p.x + margin.left;
    should_collapse := is_first && !p.is_bfc;
    uncollapsed_top :=
      if should_collapse then max(p.top, margin.top) - p.top
      else 0;
    box.y := p.y + p.h + uncollapsed_top + p.top;
    box.w := p.w - (margin.left + margin.right);
    total.h := h_acc + uncollapsed_top + margin.bottom;
    cs.p.is_bfc := is_bfc;
    cs.p.x := box.x;
    cs.p.y := box.y;
    cs.p.w := box.w;
    cs.p.h := @{h_acc};
    cs.p.top := collapsed_top;
    h_acc := fold 0 .. @{h_acc} + cs.total.h;
  }
}
```

Fig. 4. The attribute grammar that encodes the layout semantics of block elements. An attribute grammar consists of a sequence of `interface` declarations, which enumerate the available attributes, and `class` instantiations, which specify the rules for computing the attributes. Note that the computation rules are *declarative*, i.e., the user does not need to come up with an order the respects all dependencies.

sketch that specifies the high-level structure of the traversal. Here, a sketch contains holes that will be filled with computation rules. The goal of Medea is to fill in the holes to obtain layout engines that are both correct and efficient. Medea's traversal language allows the user to specify both the high-level structure of the traversal (e.g., pre-order, post-order, or hybrid) and the low-level scheduling of individual attribute computations if desired. Notably, Medea also provides traversal constructs for incrementality via `when` blocks. A `when` block encloses the computation of one of more attributes with a conditional guard, such that the enclosed attributes will only be evaluated if the guarding expression evaluates to true.

Consider our running example, the render trees for block elements. To use Medea, the user provides:

- The attribute grammar specification of the layout semantics, shown in Figure. 4. Note that the attribute grammar does not specify the order in which the attributes should be computed, nor how to "incrementalize" the computation.
- A traversal sketch (Figure. 5 (a)) that suggests the high-level structure of the traversal. In this case, the traversal first performs a `case` split based on the current node type (i.e., `CViewport` or `Block`). Each case is composed of several components: (a) ι, which denotes a *hole* that can be scheduled with computation rules from the attribute grammar; (b) `when` blocks, which denote the conditional evaluation of groups of attributes; `iterate` blocks, which iterate over a list of children and enable accesses to their attributes; (d) `recur` statements, which recursively performs the overall traversal on the specified child node.

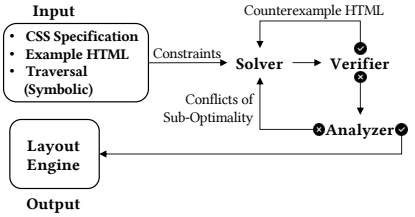

Fig. 3. Architecture of Medea.

Given the attribute grammar and the sketch, Medea searches for concrete, incremental layout engines that are derived from the sketch (e.g., Figure. 5 (b)-(c)), together with a mapping from dirty bits to attributes (e.g., Figure. 14). To ensure the proposed layout engines are both correct and efficient, Medea employs an architecture outlined in Figure. 3:

```
traversal {
  case CViewport {
    when κ0 {
      ι0;
    }
    when κ1 {
      ι1;
    }
    recur r;
    when κ2 {
      ι2;
    }
  }
  case Block {
    when κ3 {
      ι3;
    }
    when κ4 {
      ι4;
    }
    iterate cs {
      ι5;
      recur cs;
      ι6;
    }
    when κ5 {
      ι7;
    }
  }
}
```

(a) Traversal sketch

```
traversal {
  case CViewport {
    when b0 {
      eval r.p.x;
      eval r.p.w;
    }
    when b1 {
      eval r.p.y;
    }
    recur r;
    when r.b1 {
      eval total.h;
    }
  }
  case Block {
    when b0 {
      eval box.x;
      eval box.w;
    }
    when b1 {
      eval should_collapse;
      eval uncollapsed_top;
      eval box.y;
    }
    iterate cs {
      eval cs.p.{x, y, w, h};
      eval cs.p.{top, is_bfc};
      recur cs;
      eval h_acc;
    }
    when b1 || b2 {
      eval total.h;
    }
  }
}
```

(b) Candidate 1

```
traversal {
  case CViewport {
    when b0 {
      eval r.p.x;
      eval r.p.w;
    }
    when b1 {
      eval r.p.y;
    }
    recur r;
    when r.b1 {
      eval total.h;
    }
  }
  case Block {
    when b0 || b1 {
      eval box.x;
      eval should_collapse;
    }
    when b1 || b0 {
      eval uncollapsed_top;
      eval box.y;
      eval box.w;
    }
    iterate cs {
      eval cs.p.{x, y, w, h};
      eval cs.p.{top, is_bfc};
      recur cs;
      eval h_acc;
    }
    when b1 || b2 {
      eval total.h;
    }
  }
}
```

(c) Candidate 2

Fig. 5. The user of Medea provides a *sketch* as shown on (a). The sketch contains evaluation holes (i.e. ι) that can be filled with any number of attribute computation rules or dirty bits. Medea searches for concrete layout engines (e.g., the two candidates in (b) and (c)) that instantiate the sketch. Note that the second candidate may induce more *spurious updates* (i.e., it may recompute more attributes) than the first candidate, due to a less efficient scheduling of when blocks.

(1) From the specification of layout semantics as attribute grammar rules, a traversal sketch, and an initial HTML tree as an example, Medea symbolically evaluates the sketch into constraints to be solved by an off-the-shelf SMT solver [De Moura and Bjørner 2008].
(2) Next, Medea's *verifier* checks solutions for correctness over all possible HTML trees up to some depth in a standard counter-example-guided inductive synthesis (CEGIS) fashion [Solar-Lezama et al. 2006]: if the concrete layout engine is incorrect, the counterexample returned by the verifier is considered by the symbolic compilation procedure in the subsequent iterations. Note that the CEGIS loop ensures that the synthesized layout engine is correct not only on the initial HTML tree but *all* HTML trees up to some depth.
(3) To find a candidate that achieves better incremental performance, the verified layout engine is sent to Medea's *conflict analyzer*, which statically analyzes the traversal for potential sources of inefficiency. Inspired by conflict-driven SAT solvers [Silva and Sakallah 1997], the analyzer explains the root cause of inefficiency using conflict clauses that prevent the synthesizer from returning similar inefficient solutions in future iterations. The *optimal synthesis* loop (with respect to our static cost model) eventually returns an efficient layout engine.

$$
\begin{array}{rcl}
\textit{grammar} & ::= & (\ \textit{interface} \mid \textit{class}\ )^* \\
\textit{interface} & ::= & \text{interface } x\ \{\ (\ \textit{mode}\ x : Y\ )^*\ \} \\
\textit{mode} & ::= & \text{input} \mid \text{output} \\
\textit{class} & ::= & \text{class } Y : Z\ \{\ \textit{children rules}\ \} \\
\textit{children} & ::= & \text{children}\ \{\ (\ x : t\ )^*\ \} \\
t & ::= & \\
 & \mid & Z \\
 & \mid & [\ Z\ ] \\
\textit{rules} & ::= & \text{rules}\ \{\ (\ a := e;\ )^*\ \} \\
e & ::= & \\
 & \mid & f(\overrightarrow{a_i}) \\
 & \mid & c \\
 & \mid & \text{foldl } e_1\ ..\ e_2 \\
a & ::= & x \mid x\ .\ x
\end{array}
$$

$x, Y, Z, f \in$ **identifiers**

$$
\begin{array}{rcl}
\textit{prog} & ::= & \text{traversal}\ \{\ \overrightarrow{\text{case } x_i\ \{\ ss_i\ \}}\ \} \\
ss & ::= & \epsilon \mid (s;\ ss) \\
s & ::= & \\
 & \mid & \text{when}_k\ d\ \{\ es\ \} \\
 & \mid & \text{recur } x \\
 & \mid & \text{iterate } x\ \{\ ss\ \} \\
d & ::= & \text{false} \mid x \vee d \\
es & ::= & \epsilon \mid (e;\ es) \\
e & ::= & \\
 & & \text{skip} \\
 & \mid & \text{eval } a \\
 & \mid & \iota \\
a & ::= & x \mid x\ .\ x
\end{array}
$$

$x \in$ **identifiers**

Fig. 6. Syntax for the attribute grammar $\mathcal{L}_a$ and the traversal language $\mathcal{L}_t$. Derived constructs are colored in gray.

3 PROBLEM FORMULATION

In this section, we formally introduce the problem of incremental traversal synthesis. We first define the attribute grammar language by which the user can specify tree structures and computation rules (Section. 3.1). We then describe thine language for tree traversals (Section. 3.2) and define its semantics (Section. 3.3), which solvers can use to derive correct incremental traversals from sketches.

3.1 Attribute Grammar

The attribute grammar language allows the user to specify permissible tree structures and computation rules in an object-oriented style. The language is extended from Hecate [Chen et al. 2022]. Its syntax is shown in Figure. 6.

```
1  interface LayoutBox {
2    input margin.left : Pixels;
3    output p.x : Pixels;
4    output box.x : Pixels;
5    ...
6  }
7  class Block : LayoutBox {
8    children { cs : [LayoutBox]; }
9    rules {
10     box.x := p.x + margin.left;
11     ...
12   }
13 }
```

Fig. 7. Excerpt of the render tree attribute grammar from Figure. 4.

The tree structure is specified using a combination of interface and class. An interface enumerates the primitive fields, called *attributes*, that are shared by all classes instantiating said interface. An attribute can be either an input attribute, whose value is provided by the user and hence do not depend on any other attribute, or an output attribute, for which a computation rule must be provided. If an attribute is output, then it must be computed once and only once during the traversal.

A class specifies its children and the type of each children in the children section, as well as rules for computing the interface attributes in the rules section. The rules section consists of statements that specify how to compute the left-hand side (LHS) attribute using the expression on the right-hand side (RHS). The only essential aspect of a computation rule is the dependency between the LHS and the RHS attributes.

Example 3.1. Figure. 7 shows a fragment of the attribute grammar for the render tree example. It defines the `LayoutBox` interface, instantiated by `Block` class. Each node of class `Block` contains an array of `LayoutBox` children, and has the `box.x` attribute which is computed by summing the output attribute `p.x` with the input attribute `margin.left`.

3.2 Language for Incremental Tree Traversals

The syntax of the tree traversal language $\mathcal{L}_t$ is defined in Figure. 6. At the top level, a tree traversal program is a case split on the class of the current node. Each case consists of statements s which can be either a when-block when or a recursive traversal recur into a child of the current node, or iterating over an array of children. A when-block when contains a sequence of evaluation e, guarded by a dirty-bit condition expression d, which is a disjunction of boolean-valued dirty bits. An evaluation can be concrete and computes at most one attribute, or it can be a hole ι that represent a non-deterministic choice.

Definition 3.2. We say a traversal program P is *symbolic* (or, P is a *sketch*) if it contains at least one hole, and *concrete* otherwise. Furthermore, a concrete traversal P is a *completion* of the symbolic traversal P^*, if P can be obtained from P^* by filling every hole with skip or eval (a) for some a.

3.3 Semantics for Symbolic Traversal Programs

In this section, we define the semantics for evaluating symbolic traversal programs written in $\mathcal{L}_t$ in terms of the SMT constraints that each program construct generates (Figure. 8).

3.3.1 Domains and Functions. The following domains and functions are required to define the evaluation relations.

- The set $\mathbb{A}$ contains the regular attributes, and the set $\mathbb{D}$ contains special attributes that serve as the dirty bits.
- A traversal operates over a set of trees, each containing a set $\mathbb{N}$ of nodes and a distinguished node called the root node. The partial function GetChild identifies the child node of a given node by name, if it exists.
- A node n has a class c and a set of locations $\mathbb{L}$ that hold the runtime values of attributes of said node. The dereference operator $^* : \mathbb{A} \times \mathbb{N} \hookrightarrow \mathbb{L}$ returns the location associated with a valid attribute-node pair. Since each (output) attribute has exactly one computation rule for every node class, the function α returns the unique computation rule for computing each location. Moreover, the function IsInput determines whether a location is an input.
- Each traversal contains a sequence $\mathbb{I}$ of holes to be filled with computation rules, and and a sequence $\mathbb{K}$ of guard expression holes, one for every when-block.
- The time domain $t \in \mathbb{T}$ of non-negative integers is used to enforce the ordering of attribute computation.
- The function $\delta : \mathbb{A} \to \mathbb{D}$ is induced by the dirty bit mapping, which determines the dirty bit that should be set to true if an attribute is modified.

3.3.2 Predicates. Evaluating a symbolic traversal generates constraints over the following predicate unknowns, whose solutions correspond to completion of the symbolic traversal.

- $\sigma : \mathbb{I} \times \mathbb{A}$ determines whether attribute $a \in \mathbb{A}$ should be scheduled in slot $\iota \in \mathbb{I}$.
- $\gamma : \mathbb{D} \times \mathbb{K}$ determines whether dirty bit $b \in \mathbb{D}$ should appear positively in the disjunctive guard expression for when-block k.
- $\rho : \mathbb{L} \times \mathbb{T}$ determines if location l is ready (i.e., was written to) before or at time $t \in \mathbb{T}$.

3.3.3 Evaluation Relations. As shown in Figure. 8, there are three different evaluation relations:

- *Program evaluation relation:* We use judgment of the form $\langle tr;\ P\rangle \Downarrow_P C$ to evaluate traversal P on tree tr, yielding constraint C.
- *Statement evaluation relation:* The notation $\langle n, t;\ s\rangle \Downarrow_S \langle C,\ t'\rangle$ means that evaluating statement s on node n at time t yields constraint C and advances the clock to t'.

$$\frac{\langle \mathsf{root}(tr), 0;\ P\rangle \Downarrow_S \langle C,\ t\rangle \qquad C_0 = \bigwedge_{l\in\mathbb{L}, l=^*(a,n)} (\rho(l,0) \iff \mathsf{IsInput}(a) = \mathsf{true}) \wedge \rho(l,t) \qquad C_1 = \bigwedge_{a\in\mathbb{A}} \mathsf{IsInput}(a) = \mathsf{false} \implies \left(\bigvee_{\iota\in\mathbb{I}} \sigma(a,\iota) \wedge \bigwedge_{\iota\in\mathbb{I}} \bigwedge_{\iota\neq\iota'\in\mathbb{I}} \neg\left(\sigma(a,\iota)\wedge\sigma(a,\iota')\right)\right)}{\langle tr;\ P\rangle \Downarrow_P C \wedge C_0 \wedge C_1}\ \text{E-Prog}$$

$$\frac{\mathsf{ClassOf}(n) = x_i \qquad \langle n, t;\ ss_i\rangle \Downarrow_S \langle C,\ t'\rangle}{\langle n, t;\ \overrightarrow{\mathsf{case}\ x_i\ \{\ ss_i\ \}}\rangle \Downarrow_S \langle C,\ t'\rangle}\ \text{E-Case} \qquad \frac{\langle n, t_0;\ s\rangle \Downarrow_S \langle C_1,\ t_1\rangle \qquad \langle n, t_1;\ ss\rangle \Downarrow_S \langle C_2,\ t_2\rangle}{\langle n, t_0;\ (s;\ ss)\rangle \Downarrow_S \langle C_1 \wedge C_2,\ t_2\rangle}\ \text{E-SSeq}$$

$$\frac{(n,x) \in \mathsf{dom}(\mathsf{GetChild}) \qquad \mathsf{GetChild}(n,x) = n' \qquad \langle n', t;\ P\rangle \Downarrow_S \langle C,\ t'\rangle}{\langle n, t;\ \mathsf{recur}\ x\rangle \Downarrow_S \langle C,\ t'\rangle}\ \text{E-Recur} \qquad \frac{\langle n, t, k, t;\ es\rangle \Downarrow C \qquad m = \mathsf{length}(es)}{\langle n, t;\ \mathsf{when}_k\ c\ \{es\}\rangle \Downarrow_S \langle C,\ t+m\rangle}\ \text{E-When}$$

$$\frac{\langle n, t_0, k, t_w;\ e\rangle \Downarrow C_1 \qquad \langle n, t_1, k, t_w;\ es\rangle \Downarrow C_2}{\langle n, t_0, k, t_w;\ (e;\ es)\rangle \Downarrow C_1 \wedge C_2}\ \text{E-ESeq} \qquad \frac{}{\langle n, t, k, t_w;\ \mathsf{skip}\rangle \Downarrow \mathsf{true}}\ \text{E-Skip}$$

$$\frac{\begin{array}{c} l = {}^*(a,n) \qquad \alpha(l) = \mathsf{f}(\overrightarrow{a_i}) \\ \overrightarrow{l_i} = \overrightarrow{{}^*(a_i,n)} \qquad C_1 = \left(\bigwedge_i \rho(l_i,t)\right) \wedge \neg\rho(l,t) \wedge \rho(l,t+1) \qquad C_2 = \bigwedge_i \rho(l_i,t_w) \implies \gamma(k,\delta(a_i)) \\ C_3 = \bigwedge_{l'\in\mathbb{L}} l' \neq l \wedge \neg\rho(l',t) \implies \neg\rho(l',t+1) \qquad C_4 = \bigwedge_{l'\in\mathbb{L}} \rho(l',t) \implies \rho(l',t+1) \end{array}}{\langle n, t, k, t_w;\ \mathsf{eval}\ a\rangle \Downarrow C_1 \wedge C_2 \wedge C_3 \wedge C_4}\ \text{E-Eval}$$

$$\frac{\begin{array}{c} \overrightarrow{\langle n, t, k, t_w;\ \mathsf{eval}\ a_i\rangle \Downarrow C_i} \quad \text{for all } a_i \in \mathbb{A} \qquad \langle n, t, k, t_w;\ \mathsf{skip}\rangle \Downarrow C_s \\ G_i = \sigma(\iota, a_i) \implies C_i \qquad G_s = \neg\bigwedge_i \sigma(\iota, a_i) \implies C_s \qquad C = \bigwedge_{a\in\mathbb{A}} \bigwedge_{a\neq a'\in\mathbb{A}} \neg\left(\sigma(a,\iota)\wedge\sigma(a',\iota)\right) \end{array}}{\langle n, t, k, t_w;\ \iota\rangle \Downarrow \bigwedge_i G_i \wedge G_s \wedge C}\ \text{E-Hole}$$

Fig. 8. Semantics for evaluating symbolic traversals in $\mathcal{L}_t$.

- *Attribute evaluation relation:* Finally, $\langle n, t, k, t_w;\ e\rangle \Downarrow C$ generates constraint C by evaluating attribute evaluation expression e on node n at time t. In addition, the state tuple encodes the fact that the expression is placed inside the when-block k first entered at time t_w.

We assume that traversal program P and tree tr are implicit in the statement and the attribute evaluation relations.

3.3.4 Program Evaluation. The program evaluation rule (E-Prog) interprets a traversal program P on tree tr, by passing the arguments to the statement evaluation relation and setting the time to 0. It also imposes additional constraints on ρ and σ. Specifically, it uses C_0 to assert that a location is ready at time 0 if and only if the location is an input, and that all locations must be ready by the time the traversal is complete. Moreover, it imposes restriction on the meaning of the assignment operator σ, by requiring that every output attribute be scheduled into exactly one hole ι (via C_1).

Example 3.3. Consider interpreting the traversal sketch for the render tree example using the concrete tree shown in Figure. 9. Note that `vp.w` is an input location, since `Viewport.w` is declared as input in the grammar, and `r.box.x` is an output, as `LayoutBox.box.x` is declared as an output. Thus, C_0 of rule E-Prog implies that $\rho(\mathtt{vp.w}, 0) = \mathsf{true}$ whereas $\rho(\mathtt{r.box.x}, 0) = \mathsf{false}$.

3.3.5 Statement Evaluation. The statement evaluation rules are straightforward:

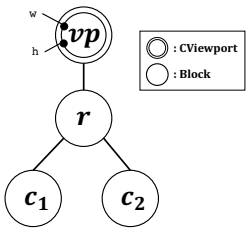

Fig. 9. An example render tree.

```
1  traversal {
2    case CViewport {
3      when .. {
4        // candidate rules:
5        // r.p.w := w (ok)
6        // total.h := r.total.h (bad)
7        ι0
8      }
9      ...
10     recur r
11     ...
12   }
13   case Block { ... }
14
```

Fig. 10. Excerpt of the traversal sketch.

```
1  traversal {
2    case CViewport { ... }
3    case Block {
4      when b0 || b1 {
5        // box.x := p.x + margin.left;
6        eval box.x;
7        // should_collapse := is_first && !p.is_bfc;
8        eval box.should_collapse;
9      }
10     ...
```

Fig. 11. Excerpt of candidate 2 from Figure. 5 showing a spurious dependency.

- E-Case performs a case analysis on the class of the current node n, and descends into the correct branch for its class.
- E-Recur tests if the current node n has a child named x, and if so, recursively performs the traversal on the child node.
- E-When interprets the attribute evaluation expressions enclosed in its body while recording the identifier of the current block k and the entry time t_w in the state. When the evaluation finishes, the time is updated with the number of attributes evaluated.

3.3.6 Attribute Evaluation. Besides sequencing, an attribute evaluation expression can be skip, eval a for some attribute a, or a hole ι. The rule E-Skip returns the trivial constraint. The rule E-Eval is more involved. On a high-level, this rule is responsible for making sure the read-write dependencies between different attributes are respected. Specifically, the following steps are performed to evaluate an attribute a:

- *Dereference.* We first dereference a on the current node n to obtain a runtime location l. The rule for computing l is retrieved to be $\mathsf{f}(\overrightarrow{a_i})$, where each a_i is dereferenced to location l_i relative to the current node.
- *Read-write dependency.* The computation of l induces read-write dependencies between itself and each l_i. To fulfill them, C_1 requires that each l_i must have, and l must have not, been computed before or at time t. Then, l becomes available at time $t + 1$.
- *Incremental dependency.* The l_i's also imply that their dirty bits must appear in the disjunctive guard expression of the current when-block k. Suppose during an incremental traversal, some l_i have been modified and its dirty bit $\delta(a_i)$ marked dirty, which means that we must recompute l. If the dirty bit do no appear positively in the guard expression, then current when-block will be skipped, violating the read-write dependencies between l_i and l. Constraint C_2 prevents this from happening by requiring $\delta(a_i)$ to appear in guard k. It also employs a small optimization that avoids assigning the dirty bit if l_i was computed in the same block.
- *Invariance.* The two lemmas C_3 and C_4 say that the availability of every location remains the same at time $t + 1$, except possibly for l.

Finally, the rule E-Hole non-deterministically expands an evaluation hole ι into a skip or an eval a_i for every attribute a_i. The result of evaluating each a_i is guarded by the pre-condition $\sigma(\iota, a_i)$ which schedules a_i into ι (via G_i). It also guards the result of evaluating skip with the pre-condition that no attribute is scheduled into ι (via G_s). Lastly, C asserts that each ι can accommodate at most one attribute.

Example 3.4. We first illustrate the E-Eval rule using the fragment of traversal sketch shown in Figure. 10. The first hole is denoted by ι_0, and can be scheduled with the computation of the following candidate attributes, among others:

- `r.p.w := w`, which computes attribute `p.w` on child `r` using the input attribute `w` of the current node;
- `total.h := r.total.h`, which computes attribute `total.h` on the current node using attribute `total.h` on child `r`.

The first candidate is valid. The E-Eval rule, if interpreted on node `vp` shown in Figure. 9, produces $C_1 = \rho(\texttt{vp.w}, 0) \wedge \neg\, \rho(\texttt{r.p.w}, 0) \wedge \rho(\texttt{r.p.w}, 1)$, which is satisfiable. The E-Eval additionally produces $C_2 = \rho(\texttt{vp.w}, 0) \implies \gamma(w_0, \delta\, \texttt{vp.h})$, i.e., if `vp.w` was not ready upon entering the `when` block, then its dirty bit δ `vp.h` must appear in the guard expression of the current block w_0.

In contrast, the second candidate is invalid: the E-Eval rule produces $C_1' = \rho(\texttt{r.box.h}, 0) \wedge \neg\, \rho(\texttt{vp.h}, 0) \wedge \rho(\texttt{vp.h}, 1)$, but because `r.box.h` cannot be determined until it has been recursively traversed, C_1 will be unsatisfiable.

Because there are more candidates for ι_0 than the two candidates shown above, the E-Hole rule expands ι_0 into the non-deterministic evaluation of at most one of the candidate attributes defined in the `CViewport` class.

We now state the problem of incremental traversal synthesis.

Definition 3.5 (Incremental tree traversal synthesis). Given an $\mathcal{L}_a$ attribute grammar L and an $\mathcal{L}_t$ traversal sketch P^*, let $\mathbb{TR}$ be the set of all trees induced by L. The incremental tree traversal synthesis problem is to find a completion P of P^* such that for every $tr \in \mathbb{TR}$, if $\langle tr;\ P\rangle \Downarrow_P C$, then C is satisfied.

3.4 Spurious Dependencies

Definition 3.5 ensures that synthesized incremental traversals are correct. However, due to the presence of dirty bits and when-blocks, some traversals will perform less unnecessary recomputation during an incremental rerun. Concretely, given a tree tr, an attribute grammar induces a dependency graph over all locations of tr. If we randomly pick a location l and modify its value, then any location that transitively depends on l needs to be recomputed. However, evaluating a traversal on tr may end up recomputing more locations than necessary, since attributes inside a when-block are updated in an all-or-none fashion.

Definition 3.6 (Spurious dependency). If location l' does not depend on l, but a modification of l causes recomputing l', then we say there is a *spurious dependency* from l to l'.

Example 3.7. Figure. 11 highlights a fragment of candidate B in Figure. 5 that contains spurious dependencies. Suppose in the dirty bit mapping, all `box.x`'s RHS attributes are mapped to `b0`, and all `box.uncollapsed_top`'s RHS attributes are mapped to `b1`. Imagine `p.x` is modified on node r, and `b0` marked `true`. The when-block will be executed since `b0` appears in the disjunction. In this case, `box.y` is unnecessarily recomputed, since `box.y` does not depend on `p.x` (or any attributes mapped to `b0`), as per the dependencies induced by the attribute grammar. Hence, we say there is a spurious dependency from `p.x` to `box.y` on node n.

We refine the goal of incremental synthesis to be finding traversals that also minimize the number of spurious dependencies. However, spurious dependency is a runtime notion: it depends on both the concrete tree being traversed, and the initial location modified. In the next section, we will develop a static approximation of spurious dependencies, and an algorithm to search for efficient traversals in terms of a static cost model.

4 CONFLICT-DRIVEN TREE TRAVERSAL SYNTHESIS

In this section, we introduce our conflict-driven algorithm for incremental tree traversal synthesis. In what follows, we first give a high-level overview of the algorithm, after which we expand on each major component of the algorithm in turn.

4.1 Overview

Given Definition 3.5, a naive solution is to encode the whole synthesis problem as a single SMT formula and send it to an off-the-shelf solver. However, it is difficult to come up with an objective function that precisely captures dynamic spurious dependencies that lead to efficiently solvable constraints. To address this challenge, we design a conflict-driven synthesis algorithm that iteratively searches for better candidates guided by a *static cost model*. Section. 4 shows the high-level structure of the synthesis algorithm. Given an attribute grammar L, a traversal sketch P^*, and an example tree tr, it attempts to find a correct completion P of sketch P^* that minimizes spurious dependencies.

Algorithm 1 Conflict-Driven Layout Engine Synthesis

```
   Input: Attribute grammar L, sketch P*, and example tree tr
   Returns: A concrete traversal P, or ⊥
1: procedure Synthesize(L, P*, tr)
2:     Ω ← ∅
3:     P◇ ← ⊥
4:     while true do
5:         C ← ⟨tr; P*⟩ ⇓_P C
6:         P ← Solve(SymCompile(C) ∧ Ω)
7:         if P = UNSAT then return ⊥
8:         else if timeout then return P◇
9:         else if Cost(P) < Cost(P◇) then P◇ ← P
10:        end if
11:        Ω ← Ω ∪ AnalyzeConflicts(P)
12:    end while
13: end procedure
```

The core of the algorithm is a conflict-driven loop. Internally, the algorithm maintains a knowledge base Ω with conflict clauses (line 2) learned from past iterations of the synthesis loop. Intuitively, conflict clauses prevent the solver from revisiting a large space of sub-optimal candidate traversals. Initially, Medea leverages the operational semantics in Section. 3 (line 5) to compile the traversal sketch into SMT constraints. It invokes the SymCompile procedure (Section. 4.2) to make the constraints efficiently solved by an off-the-shelf solver (line 6). If the constraints can be solved into a concrete traversal P, the algorithm updates the current candidate $P^\diamond$ with one that has a lower cost (line 9). Afterwards, the algorithm calls the AnalyzeConflicts routine (line 11) to examine the candidate for potential sources of inefficiency due to spurious dependencies. The conflict analysis (Section. 4.4) provides the knowledge base Ω with a set of *conflicts clauses* that explain the cause of inefficiency and can be used to block a large space of candidate traversals that are sub-optimal due to similar reasons [Feng et al. 2018]. Once timed out, the algorithm returns the traversal encountered so far that has the lowest cost.

4.2 Domain-Specific Symbolic Compilation

The SymCompile procedure leverages domain-specific symbolic compilation [Chen et al. 2022] to obtain constraints that are more efficiently solvable. Recall (Section. 3.3.2) that the constraint set C generated by the evaluation relation $\Downarrow P$ contains three predicates: σ (assigning attributes to holes), γ (assigning dirty bits to guard expressions), and ρ (modeling whether a location has been written to at each time step). Essentially, SymCompile transforms C into an equivalent C' that no longer contains the ρ predicate. The key insight is that $\rho(l, t)$, which indicates whether location $l = n.a$ for some node n and attribute a, corresponds to whether there exist a hole ι and time $t_0 \leq t$ such

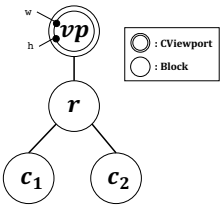

Fig. 12. The exxample render tree from Figure. 9.

```
traversal {
  case Viewport { ... }
  case Block {
    when .. { ι0; }
    when .. { ι1; }
    iterate cs { ... }
    when .. { ι2; }
}
```

Fig. 13. An excerpt of the traversal sketch for the motivating example.

that $\sigma(\iota, a)$ holds. This allows SymCompile to replace constraints involving $\rho(l, t)$ with ones that involve $\sigma(\iota, a)$[4].

Example 4.1. We illustrate the SymCompile procedure using the traversal sketch shown in Figure. 13. Suppose we are currently on node `r` (shown in Figure. 12) at time step t, and are examining the ι_2 hole in the `Block` case. Assume E-Hole has chosen to expand this hole into eval `total.h`, which requires `uncollapsed_top` to have been computed. Then, one of the constraints we will obtain is $C = \sigma(\iota_2, \texttt{total.h}) \implies \rho(\texttt{r.uncollapsed\_top}, t)$, i.e., if `total.h` is assigned to ι_2, then the value of its dependency `r.uncollapsed_top` should be ready at time t. Note, however, that `r.uncollapsed_top` can only be ready if it has been written to, i.e., if eval `uncollapsed_top` was assigned to a previous hole. Thus, SymCompile transforms C into an equivalent constraint that no longer mentions ρ:

$$C' \equiv \sigma(\iota_2, \texttt{total.h}) \implies (\sigma(\iota_0, \texttt{uncollapsed\_top}) \vee \sigma(\iota_1, \texttt{uncollapsed\_top}))\,.$$

4.3 Static Spurious Dependencies and Cost Model

Section 3 introduces spurious dependencies to quantify how much unnecessary recomputation will be performed by an incremental schedule. In practice, a spurious dependency depends on not only a concrete tree, but also an initial set of modified locations. Therefore, directly optimizing over the precise spurious dependency is computationally prohibitive. Thus, in what follows, we leverage an static approximation, called *static spurious dependencies*, which emerge from the *attribute dependency graphs*.

Definition 4.2. An attribute dependency graph $G = (V, E)$ is a graph where

- the vertex set $V = \mathbb{A} \cup \mathbb{D}$ includes the attributes and the dirty bits, and
- the edge set $E = E_c \cup E_d$ contains data- (E_d) and control-dependent (E_c) edges among attributes and dirty bits. There is a data-dependency edge (x, y) if (a) there exists an rule in which x appears in the RHS expression, and y is the LHS, or (b) x is the dirty bit for y. There is a control-dependency edge if there is a when-block in which x appears in the guard expression, and y appears as a LHS in the body.

We use the notation G_d to refer to G with edge set restricted to E_d, and G_c with edge set restricted to E_c.

Definition 4.3. Given G_c and G_d, the *static spurious dependency graph* G' is computed as $G' = \overline{G_c} \setminus \overline{G_d}|_{\mathbb{D} \cup \mathbb{A}_w}$. That is, G' is obtained by first taking graph difference between the transitive closure of G_c with the transitive closure of G_d, and then restricting the vertex set to $\mathbb{D} \cup \mathbb{A}_w$, where $\mathbb{D}$ is the set of dirty bits, and $\mathbb{A}_w$ is the set of attributes that are written to in a when-block.

[4]In practice, the constraints produced by SymCompile can be encoded as integer linear programming (ILP) constraints to further improve the solving speed. Here, we stick to the boolean constraint form to simplify the presentation.

(a) Attribute grammar

```
class Block : LayoutBox {
  children { cs : [LayoutBox]; }
  rules {
    box.x := p.x + margin.left;
    should_collapse := is_first && !p.is_bfc;
```

(b) Dirty-bit map

Class	Bit	Attributes
CViewport	b0	w
CViewport	b1	h; total.h
Block	b0	margin.{left,right}; p{x.w}; box.{x,w}
Block	b1	margin.{top,bottom}; p.{t,h,top,is_bfc}; box.y; should_collapse; uncollapsed_top; is_bfc; is_first
Block	b1	h_acc;total.h

(c) Traversal

```
case Block {
  when b0 || b1 {
    // box.x := p.x + margin.left;
    eval box.x;
    // should_collapse := is_first && !p.is_bfc;
    eval box.should_collapse;
```

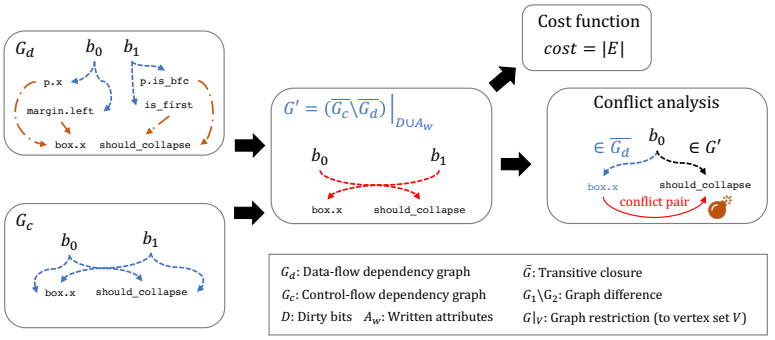

Fig. 14. Illustration of the static cost model and conflict analysis based on the attribute dependency graphs.

Example 4.4. The graphs G_d, G_c in Figure. 14 constitutes the attribute dependency graph, constructed from the attribute grammar (a), the dirty bit map (b), and the traversal (c). The data-flow dependency graph G_d represents true dependencies: black edges indicate dependencies induced by the attribute grammar, and blue edges indicate dependencies induced by the dirty bit mapping. The control-flow dependency graph G_c introduces potentially spurious dependencies. In this example, it is obtained by connecting `b0` and `b1` to each attribute that is being written to in the block. Then, we obtain static spurious dependency G' using the restricted difference of the transitive closures of the two graphs.

Definition 4.5. A cost model $\text{Cost}(P)$ is a function that maps P to its corresponding static cost by summing up the number of static spurious dependencies of all blocks in P. Thus, the cost of P is the cardinality of edge set of the static spurious dependency graph $G' = (V', E')$:

$$\text{Cost}(P) = |E'|$$

Example 4.6. The `when`-block shown in Figure. 14 (a) incurs a static cost of 2, since there is a static spurious dependency from `b1` to `box.x`, and another from `b0` to `should_collapse`.

4.4 Conflict-Driven Clause Learning

Once MEDEA synthesizes a concrete schedule P, one naive way for getting another schedule P' with better performance is to block the current model P, and keep invoking the solver until we get a better version. With static spurious dependencies and cost model, we can now blame certain bad scheduling decisions made by the solver. Importantly, the synthesizer can effectively avoid a class of similar scheduling decisions that would otherwise contribute to the same kind of static spurious dependency by using conflict clauses (added to the knowledge base) to explain the root cause of the static spurious dependencies.

To explain the root cause of a static spurious dependency, recall that a static spurious dependency from dirty bit b to attribute a arises, if a is not reachable from b in the dirty-bit-attribute dependency graph G_2 in Figure. 14. However, as b is synthesized due to the presence of some source attribute s, we know the when-block must also compute some attribute a' that depends on s and shares no common dependencies with a. If we can schedule attribute a into a different block from any such a', the static spurious dependent edges may be eliminated. Therefore, the root cause of the static spurious dependency is the co-existence of both a and a' in the current when-block. Figure. 14 shows the core idea of ANALYZECONFLICTS(P) procedure.

Let E' denote the edge set for the spurious dependency graph. Let $(d, a) \in E'$ be a spurious static dependency edge from dirty bit d to attribute a. For every written attribute a' that shares the same

when-block with a, and has a data dependency with dirty bit d according, Medea introduces a *conflict clause* that schedules the computation of a into a different block from a'. Concretely, if $\mathbb{I}_w$ is the set of available slots in the current `when` block, then the constraint

$$\bigvee_{\iota \in \mathbb{I}_w} \sigma(\iota, a') \implies \neg \bigvee_{\iota \in \mathbb{I}_w} \sigma(\iota, a)$$

asserts that a should not be scheduled inside the current when-block if a' is present.

Example 4.7. In the conflict analysis of Figure. 14, consider the static spurious dependency from `b1` to `box.x`, which can be explained by the co-existence of `box.x` and `should_collapse` in the same when-block. Assuming that the current when-block has slots ι_0 and ι_1, then the conflict is encoded as the following constraint:

$$\sigma(\iota_0, \texttt{should\_collapse}) \vee \sigma(\iota_1, \texttt{should\_collapse}) \implies \neg(\sigma(\iota_0, \texttt{box.x}) \vee \sigma(\iota_1, \texttt{box.x})).$$

5 IMPLEMENTATION

This section discusses the design and implementation details of Medea.

Traversal sketch generation. When the user provides the symbolic traversal, it might difficult to know beforehand the most efficient arrangement of when-blocks. To address this, Medea lets the user provide a *meta sketch* that fixes the high-level structure of the traversal without specifying concrete placement of incremental constructs, as well as a granularity parameter W that controls the maximum number of when-blocks that can be used in each visitor. The syntax of the meta sketch is the traversal language (Figure. 6) augmented with *meta slots*, written as $\square$, that can be expanded into any number of symbolic when-blocks. From the meta sketch, Medea uses autotuning to explore all symbolic traversals satisfying the granularity parameter W, and returns a concrete traversal with the lowest cost.

Integration with browser engines. To measure the end-to-end effectiveness of Medea, we integrated Medea with the Robinson browser engine [Brubeck 2021], a proof of concept prototype written by core members from the Mozilla layout engine team. We implemented a syntax-guided code generator that translates the traversal in our tree language into the layout traversal in Robinson written in Rust.

Subtree skipping. Medea's synthesized traversals avoid unnecessary recomputation of individual attributes. Another aspect of incremental layout is *subtree skipping*: if an incremental change does not affect any attribute in an entire subtree, then an incremental traversal does not need to visit the subtree. Although Medea's core traversal language does not include subtree skipping, an attribute-skipping traversal can be easily adapted into a subtree-skipping traversal during the code generation phase. Specifically, we assume that the browser maintains a `damage` bit for each node n, which indicates whether node n bears the initial incremental change, or n is an ancestor of such a node. We then syntactically guard every `recur` to children such that the recursive traversal happens if the child's `damage` bit is true or any of the child's inherited attributes is dirty.

6 EVALUATION

In this section, we describe the results for the experimental evaluation, which is designed to answer the following key research questions:

- **RQ1:** Can Medea synthesize fast incremental traversals, and do so efficiently?
- **RQ2:** What is the impact of Medea's conflict analysis procedure? In particular, how does it affect synthesis efficiency as well as the performance of synthesized traversals?
- **RQ3:** Is Medea's tree language expressive? In particular, can it express important CSS constructs as attribute grammar and synthesize correct traversals that bypass layout bugs from real-world layout engines?

- **RQ4:** Can one rapidly adapt Medea according to changes in the specification?

6.1 Performance of Synthesized Traversals

In this section, we compare the performance of the incremental traversals synthesized by Medea against various non-incremental and incremental baselines.

Benchmarks. Due to the lack of standardized benchmarks for evaluating incremental performance, we run candidate traversals on randomly constructed DOM trees that resemble realistic workload faced by real-world layout engines. Consistent with common HTML documents, each DOM tree contains between 200 and 4000 nodes, averaging 1000 nodes per tree.

Experiment set-up. We supply Medea with an attribute grammar that incorporates fundamental CSS layout constructs using the block layout (see Section. 6.3 for a description of features encoded in our attribute grammar). We use the code generator of Medea to plug the synthesized traversals into the Robinson web layout engine [Brubeck 2021]. Synthesis as well experiments are performed on a MacBook Pro with Intel Core i5 CPU and 16GB of RAM.

Variants. We compare the following variants of incremental traversal in our experiment:

- Hecate is a traversal synthesized by [Chen et al. 2022] using our attribute grammar. However, Hecate cannot synthesize incremental traversals, so Hecate always visits every node and recomputes every attribute.
- Hecate-**Inc** is a naive incremental traversal mechanically derived from Hecate: it adds one dirty bit per attribute and guards every attribute computation rule with a precise dirty bit condition. As a result, this variant theoretically minimizes the number of spurious updates.
- Medea is the incremental traversal synthesized by our tool using a 60-minute timeout.

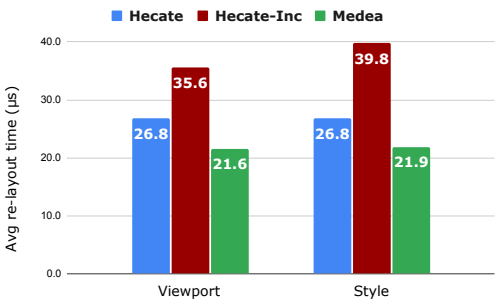

Fig. 15. Average re-layout time for three traversals: Hecate, Hecate-**Inc**, and Medea due to an incremental change to viewport dimensions or style of the root node.

In the experiments, we first use Hecate to perform the initial layout. Then we simulate two kinds of incremental changes that potentially affect all nodes in a layout tree: 1) resizing the browser viewport to random dimensions, and 2) randomly modifying an CSS attribute of the root element. Finally, we run each of the three variants to perform a re-layout and measure the running time, averaged over 1000 trials.

Figure. 15 compares the average re-layout time of each variant. In particular, Medea is 1.24× faster than Hecate, and 1.64× faster Hecate-**Inc** for incremental viewport resizing. For incremental style modification, Medea is 1.22× faster than Hecate, and 1.82× faster than Hecate-**Inc**. Observe that the naive strategy of turning non-incremental traversals to incremental traversals (i.e., Hecate-**Inc**) has worse performance than its non-incremental version due to the overhead of (1) maintaining the large number of dirty bits, and (2) branch misprediction penalty resulting from the excessive number of conditional statements.

Result for RQ1: Medea synthesizes incremental traversals that are 1.64–1.82× faster than a naive incremental traversal, and 1.22–1.24× faster than a non-incremental traversal.

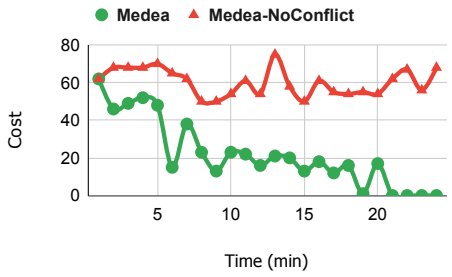

Fig. 16. Evolution of the static costs of the candidate traversals proposed by Medea and Medea-**NoConflict**. Lower is better.

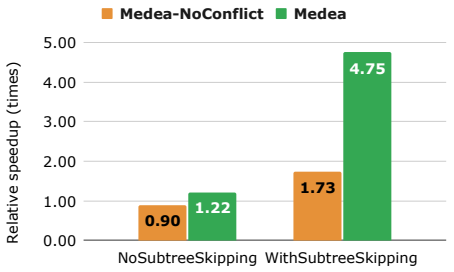

Fig. 17. Comparison of Medea and Medea-**NoConflict** in terms of traversal speedup over the non-incremental traversal baseline.

6.2 Impact of Conflict Analysis

In this section, we study the impact of Medea's conflict analysis procedure. We compare Medea against Medea-**NoConflict** replaces Medea's AnalyzeConflicts procedure with one that simply blocks all candidate traversals encountered so far, using the boolean negation of previously seen models. Essentially, Medea-**NoConflict** performs an enumerative search over valid traversals, and returns the lowest-cost traversal upon timeout.

In Figure. 16, we show the static cost (described in Section 4.1) of the candidate traversals proposed by Medea and Medea-**NoConflict** within a 25-minute time period. Because Medea learns conflict clauses that prevent future candidates from having the same source of inefficiency, overall it is able to propose increasingly lower-cost traversals as time evolves. In contrast, the candidates proposed Medea-**NoConflict** remain costly as it does not learn from past mistakes. On average, Medea identified 33 conflicts per iteration.

To compare the end-to-end performance of the best traversals synthesized by Medea and Medea-**NoConflict**, we run each tool using a 60-minute timeout and use the lowest-cost traversal found by each tool. We then perform a random incremental change to a randomly chosen node for each HTML benchmark page, and measure the re-layout time using the incremental traversal synthesized by each tool compared to a non-incremental baseline. Figure. 17 shows the relative speedup of each tool. With subtree skipping disabled, Medea achieves a 1.22× speedup over the non-incremental baseline, compared to a 10% slowdown from Medea-**NoConflict**. With subtree skipping enabled, the performance gap increases, where Medea is 2.74× faster than Medea-**NoConflict**.

> **Result for RQ2:** Medea's conflict analysis learns from past mistakes, which leads to efficient layout engines compared to a naive synthesis approach. Medea synthesizes incremental traversals that are 2.74× faster than traversals synthesized without conflict analysis.

6.3 Correctness in Bypassing Real-World Layout Bugs

In addition to obtain high-performance layout engines through incrementality, we also need to ensure that the generated layout engines are correct. To evaluate whether the layout engine synthesized by Medea can bypass real-world bugs, we consolidate a list of 31 layout engine bugs from mainstreaming browsers, namely, Firefox, Chrome, and Safari. [5] Each benchmark in our data set needs to satisfy two requirements: 1) the CSS rules that are relevant to the bug are expressible in our attribute grammar, and 2) the original bug contains at least one test case.

[5]We use Firefox, Chrome, and Safari to represent their corresponding layout engines, namely, Servo, Chromium, and Webkit.

	CSS 2.0	CSS 2.1	CSS 2.2
Specification (LOC)	384	439 (Δ = 59)	433 (Δ = 68)
Implementation (LOC)	1398	1413 (Δ = 112)	1436 (Δ = 136)

Fig. 18. Three versions of our CSS semantics, differing in their margin collapsing implementation (as specified in CSS 2.0, 2.1, and 2.2 [Bos 2016; Bos et al. 2011, 1998]). The table gives line of code and changes from the previous version.

Summary of results. The layout engine synthesized by Medea is able to generate the correct layouts for all benchmarks except for one (10151). The cause of that benchmark is due to an unstandardized feature in margin collapse and browsers interpret it differently [Mozilla 2019]. Details of the results are provided in the Appendix.

Result for RQ3: The language of Medea can express complex CSS semantics, and Medea can synthesize a layout engine that is able to bypass many real-world layout bugs.

6.4 Adaptability to Specification Changes

Browser layout engines are in constant flux, adapting to changes in the specification and implementing newly standardized features. For example, in 2017, the tracking website CanIUse.com lists 10 features newly implemented in Firefox and 11 in Chrome [caniuse 2019], ranging from `display: flow-root` to Grid Layout Level 1. Medea assists browser developers with adapting their browsers to changing requirements: by synthesizing a correct and efficient layout engine from a high-level specification, developers using Medea avoid splitting their attention between correctness and performance.

Consider the differences in margin collapsing behavior between CSS versions 2.0, 2.1, and 2.2 (draft) [Bos 2016; Bos et al. 2011, 1998]. Between version 2.0 and 2.1, for example, elements with zero height and clearance stopped collapsing with their parent's bottom margin; between 2.1 and 2.2, elements with non-zero minimum height but automatically computed height stopped collapsing with their parent's bottom margin, but only if the minimum height isn't binding. Browser developers must adapt to these changes in specification by modifying their layout engines, adding additional state and modifying the sequence of layout passes. Furthermore, due to the complex dependencies between different features (margin collapsing, for example, affects height computation), changes to one feature could require global changes to the structure of the full layout engine. Bugs in existing layout engines are often left unfixed due to the difficulty of executing the global changes required to implement fixes [Eklund 2019].

As a case study of adapting browsers to changing specifications, we implemented three versions of our margin-collapsing specification, corresponding to the CSS 2.0, 2.1, and 2.2 semantics. The three specifications were similar, differing in only a few lines of code, and layout engines corresponding to each could be synthesized quickly (see Figure. 18). Despite the similarity of the specification, the actual implementation code differed more dramatically, with many more lines of code changed.

Result for RQ4: Medea readily adapts to changes in specification across CSS 2.0, 2.1, and 2.2.

6.5 Discussion

Coverage of CSS/HTML rules. Although Medea only supports a fragment of CSS/HTML, it covers all tricky features from the Cassius framework [Panchekha and Torlak 2016], including absolute position, margin collapses, and floating layout. Extending Medea to fully support all CSS is technically feasible but requires a significant amount of engineering effort, which is orthogonal to the core research idea of this project.

Web pages selection. Although we have implemented several challenging CSS features such as margin collapses and floating layout, there are other features (e.g., grid, flex, etc.) and javascript code that are common in real world web pages whose features are not fully ported by Medea. Thus, we created random web pages that only use the subset that is currently supported. We believe supporting most of the common CSS is possible with additional engineering effort.

Scalability of page loading time. As we mentioned earlier, a layout engine is only synthesized for once and is used by all pages. Therefore, the page load time generally scales linearly with the number of attributes in the underlying attribute grammar. However, with good subtree skipping (which depends on the traversals having few spurious computations), layout time is orders of magnitude faster than executing Javascript. Thus, even after adding more attributes to support a wider subset of CSS, the page load time should not increase significantly.

7 RELATED WORK

General tree traversal synthesis. Medea builds on Hecate [Chen et al. 2022], which performs automated tree traversal synthesis using domain-specific symbolic compilation. By reducing synthesis problems to constraint solving, Hecate optimizes upon the final constraint system to improve scalability. As Hecate can also be used for synthesis of layout engine, Medea offers more flexibility on incrementality of the synthesized traversal besides correctness. Medea can considerably explores more optimized solutions with better performance.

Grafter [Sakka et al. 2019] is another synthesizer for general tree traversals. Unlike Medea, Grafter is based on static analysis: it generates automata that capture the dependencies induced from traversal statements, and uses a deterministic algorithm to rewrite and fuse traversals into more compact ones. While Grafter is fast, extending it to new specifications may require extra expert knowledge to devise new tree fusion theories. Neither Hecate nor Grafter supports incrementality, and has no notion of optimality for their synthesized traversals.

Browser layout engines. There is a series of works that express and synthesize layouts using constraint-based systems [Badros et al. 1999; Borning et al. 1997; Hashimoto and Myers 1992; Sutherland 1964; van Wyk 1982; Zanden and Myers 1991], represent structured graphics using domain-specific languages [Wilkinson 2005], and manipulate SVG [Chugh et al. 2016]. Among them, Medea adopts the layout semantics from Cassius [Panchekha and Torlak 2016], which together with VizAssert [Panchekha et al. 2018] extends a subset of layout formalization with finitization reductions to support more CSS standards. Meanwhile, the Cornipickle project [Hallé et al. 2015] represents visual properties using first-order modal logic, which was adapted by VizAssert.

There is a rich class of tools for detecting and repairing layout issues [Bigham 2014; Mahajan et al. 2018a, 2017, 2018b; Walsh et al. 2017, 2015]. In particular, an important subclass of those tools aims to detect inconsistent rendering results between different browsers [Choudhary et al. 2012; Mesbah and Prasad 2011; Roy Choudhary et al. 2010]. While those tools are designed for web developers (whereas Medea targets browser developers), their number hints at the challenges of layout bugs and performance faced by practitioners and at the importance of the problems Medea addresses. In fact, practitioners commonly test their web pages against specific instances of browsers and operating systems by loading pages in virtual machine instances [Browserling 2018; Browsershots 2018; Browserstack 2018]. In contrast, Medea aims to synthesize optimal layout engines with incrementality support in addition to correctness, thus reduces the frequency of manual testing.

Constraint solving. Constraint solving based on Satisfiability Modulo Theories [Nelson and Oppen 1980] has become a powerful tool for symbolic reasoning as practical, high-performance solvers have become available [Barrett et al. 2011; De Moura and Bjørner 2008; Gurobi Optimization 2019].

Solver-based verification and synthesis tools have been extensively studies by the programming languages community [Leino 2010; Schkufza et al. 2013; Solar-Lezama 2008]. Traditional SMT-based tools use a custom constraint solver or manually translate problems into constraints for a specific existing solver. Solver-aided domain-specific languages [Torlak and Bodik 2013; Uhler and Dave 2014] instead automatically generate solver constraints based on symbolic compilation. For example, the Rosette framework [Torlak and Bodik 2014] uses Racket's meta-programming features to provide a high-level interface to several solvers. Medea is built on top of Rosette, but leverages Hecate's domain-specific compilation strategy to generate efficient constraints that lead to significant improvement in solving time.

Attribute grammar. Many attribute grammar formalism [Knuth 1968] assume dynamic scheduling, in contrast to the fully static scheduling presented in Medea. Unlike exiting work [Chen et al. 2022; Meyerovich et al. 2013] that leverages attribute grammars to model simple attribute rules, Medea significantly extends prior traversal languages by supporting complex features in browser layout engines such as margin collapse, floating layout, absolute positioning, and incrementality.

8 CONCLUSION

We propose Medea, a synthesis-powered framework for building layout engines with correctness guarantee while supporting incrementality, which is a crucial yet error-prone module that enables good performance in modern layout engines. We demonstrate Medea on a fragment of HTML layout covering sophisticated features like margin collapse, floating layout, and absolute positioning. Medea successfully synthesizes a layout engine for this fragment. The incremental layout engine synthesized by Medea achieves up to 1.82× speed-up compared to a naive incremental baseline. We also demonstrate the effectiveness of our conflict-driven algorithm in that it leads to 2.74× faster schedules compared to a baseline without conflict analysis.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers from both the current and the past submissions of this paper for their invaluable feedback. This work was partially funded by NSF awards SaTC–1908494, ITE–2132318, CCF–2122950, ITE–2029457, ITE–1936731, CCF–1918027, and IIS–1924435, grants from DARPA HARDEN and FA8750–16–2–0032, the Google Faculty Research Award, the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF–1723352), the CONIX Research Center (one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741–394324 AM01), as well as gifts from Adobe, Facebook, Google, Intel, and Qualcomm.

REFERENCES

Greg J. Badros, Alan Borning, Kim Marriott, and Peter J. Stuckey. 1999. Constraint Cascading Style Sheets for the Web. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST'15)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/320719.322588

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177. http://dl.acm.org/citation.cfm?id=2032305.2032319

Jeffrey P. Bigham. 2014. Making the Web Easier to See with Opportunistic Accessibility Improvement. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. ACM, New York, NY, USA, 117–122. https://doi.org/10.1145/2642918.2647357

Alan Borning, Richard Lin, and Kim Marriott. 1997. Constraints for the Web. In *Proceedings of the Fifth ACM International Conference on Multimedia* (Seattle, Washington, USA) *(MULTIMEDIA '97)*. ACM, New York, NY, USA, 173–182. https://doi.org/10.1145/266180.266361

Bert Bos. 2016. CSS 2.2: Collapsing Margins. https://tinyurl.com/j66mfru.

Bert Bos, Tantek Çelik, Ian Hickson, and Håkon Wium Lie. 2011. CSS 2.1: Collapsing Margins. https://tinyurl.com/rspsl2j.

Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. 1998. CSS 2: Collapsing Margins. https://tinyurl.com/seb5h92.

Browserling. 2018. https://www.browserling.com/

Browsershots. 2018. http://browsershots.org/

Browserstack. 2018. https://www.browserstack.com/screenshots

Matt Brubeck. 2014. Incorrect layout of element following a float, involving margins. https://github.com/servo/servo/issues/4307.

Matt Brubeck. 2021. The Robinson Layout Engine. https://github.com/mbrubeck/robinson.

caniuse. 2019. Compare Firefox 51, Firefox 58, Chrome 56, and Chrome 64. https://caniuse.com/#compare=firefox+51,firefox+58,chrome+56,chrome+6&compare_cats=CSS.

Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodik. 2022. *Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation*. Association for Computing Machinery, New York, NY, USA, 1030–1042. https://doi.org/10.1145/3503222.3507751

S. R. Choudhary, M. R. Prasad, and A. Orso. 2012. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 171–180. https://doi.org/10.1109/ICST.2012.97

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 341–354. https://doi.org/10.1145/2908080.2908103

Leonardo De Moura and Nikolaj Bjørner. 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* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Emil Eklund. 2019. LayoutNG. https://www.chromium.org/blink/layoutng.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 420–435.

Github. 2013. Electron | Build cross platform desktop apps with JavaScript, HTML, and CSS. https://electronjs.org/.

LLC Gurobi Optimization. 2019. Gurobi Optimizer Reference Manual. http://www.gurobi.com

Sylvain Hallé, Nicolas Bergeron, Francis Guerin, and Gabriel Le Breton. 2015. Testing Web Applications Through Layout Constraints. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, IEEE, Graz, Austria, 1–8.

Osamu Hashimoto and Brad A. Myers. 1992. Graphical Styles for Building Interfaces by Demonstration. In *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology* (Monteray, California, USA) *(UIST '92)*. ACM, New York, NY, USA, 117–124. https://doi.org/10.1145/142621.142635

Anantha Keesara. 2007. Bug 15662 - layout is coming down because of style 'float:left' of . https://bugs.webkit.org/show_bug.cgi?id=15662.

Donald E. Knuth. 1968. Semantics of Context-Free Languages. In *In Mathematical Systems Theory*. 127–145.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370. http://dl.acm.org/citation.cfm?id=1939141.1939161

Sonal Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G.J. Halfond. 2018a. Automated Repair of Mobile Friendly Problems in Web Pages. In *International Conference on Software Engineering (ICSE 2018)*. ACM, 140–150.

Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. Automated Repair of Layout Cross Browser Issues Using Search-based Techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/3092703.3092726

S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond. 2018b. Automated Repair of Internationalization Presentation Failures in Web Pages Using Style Similarity Clustering and Search-Based Techniques. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 215–226. https://doi.org/10.1109/ICST.2018.00030

A. Mesbah and M. R. Prasad. 2011. Automated cross-browser compatibility testing. In *2011 33rd International Conference on Software Engineering (ICSE)*. 561–570. https://doi.org/10.1145/1985793.1985870

Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, and Rastislav Bodik. 2013. Parallel Schedule Synthesis for Attribute Grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2442516.2442535

Mozilla. 2019. Margins collapse on body element that creates a BFC. https://tinyurl.com/v2284y6

Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (April 1980), 356–364. https://doi.org/10.1145/322186.322198

Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying that web pages have accessible layout. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 1–14.

Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 181–194. https://doi.org/10.1145/2983990.2984010

Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/ICSM.2010.5609723

Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, fine-grained traversal fusion for heterogeneous trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 830–844.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

João P. Marques Silva and Karem A. Sakallah. 1997. GRASP—a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design* (San Jose, California, USA) *(ICCAD '96)*. IEEE Computer Society, USA, 220–227.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGARCH Comput. Archit. News* 34, 5 (oct 2006), 404–415. https://doi.org/10.1145/1168919.1168907

stshine. 2016. Floats fails to interact with inline elements in the simplest case. https://github.com/servo/servo/issues/13683.

Ivan E. Sutherland. 1964. Sketch Pad a Man-machine Graphical Communication System. In *Proceedings of the SHARE Design Automation Workshop (DAC '64)*. ACM, New York, NY, USA, 6.329–6.346. https://doi.org/10.1145/800265.810742

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. ACM, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

Richard Uhler and Nirav Dave. 2014. Smten with Satisfiability-based Search. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 157–176. https://doi.org/10.1145/2660193.2660208

Christopher J. van Wyk. 1982. A High-Level Language for Specifying Pictures. *ACM Trans. Graph.* 1, 2 (April 1982), 163–182. https://doi.org/10.1145/357299.357303

W3C. 2007. CSS basic box model. http://www.w3.org/TR/css3-box

Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. 2017. Automated Layout Failure Detection for Responsive Web Pages Without an Explicit Oracle. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 192–202. https://doi.org/10.1145/3092703.3092712

T. A. Walsh, P. McMinn, and G. M. Kapfhammer. 2015. Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 709–714. https://doi.org/10.1109/ASE.2015.31

Patrick Walton. 2014. Block formatting contexts should never collapse margins. https://github.com/servo/servo/issues/10449.

Leland Wilkinson. 2005. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Brad Vander Zanden and Brad A. Myers. 1991. The Lapidary Graphical Interface Design Tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New Orleans, Louisiana, USA) *(CHI '91)*. ACM, New York, NY, USA, 465–466. https://doi.org/10.1145/108844.109005

Received 2022-11-10; accepted 2023-03-31