Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation

Yanju Chen
yanju@cs.ucsb.edu
University of California, Santa Barbara
USA

Junrui Liu
junrui@cs.ucsb.edu
University of California, Santa Barbara
USA

Yu Feng
yufeng@cs.ucsb.edu
University of California, Santa Barbara
USA

Rastislav Bodik
bodik@cs.washington.edu
University of Washington
USA

ABSTRACT

Efficient computation on tree data structures is important in compilers, numeric computations, and web browser layout engines. Efficiency is achieved by statically scheduling the computation into a small number of tree traversals and by performing the traversals in parallel when possible. Manual design of such traversals leads to bugs, as observed in web browsers. Automatic schedulers avoid these bugs but they currently cannot explore a space of legal traversals, which prevents exploring the trade-offs between parallelism and minimizing the number of traversals.

We describe Hecate, a synthesizer of tree traversals that can produce both serial and parallel traversals. A key feature is that the synthesizer is extensible by the programmer who can define a template for new kinds of traversals. Hecate is constructed as a solver-aided domain-specific language, meaning that the synthesizer is generated automatically by translating the tree traversal DSL to an SMT solver that synthesizes the traversals. We improve on the general-purpose solver-aided architecture with a *scheduling-specific symbolic evaluation* that maintains the engineering advantages solver-aided design but generates efficient ILP encoding that is much more efficient to solve than SMT constraints.

On the set of Grafter problems, Hecate synthesizes traversals that trade off traversal fusion to exploit parallelism. Additionally, Hecate allows defining a tree data structure with an arbitrary number of children. Together, parallelism and data structure improvements accelerate the computation 2× on a tree rendering problem. Finally, Hecate's domain-specific symbolic compilation accelerates synthesis 3× compared to the general-purpose compilation to an SMT solver; when scheduling a CSS engine traversal, this ILP-based synthesis executes orders of magnitude faster.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507751

KEYWORDS

symbolic compilation, program synthesis, tree traversal

ACM Reference Format:
Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodik. 2022. Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507751

1 INTRODUCTION

Traversal of tree structures is the foundation behind many applications: compilers leverage traversals of abstract syntax trees (ASTs) to analyze and optimize source codes. Layout engines in web browsers rely on traversals of render trees to determine the locations and appearances of HTML elements on web pages. Implementing tree traversals is a daunting task as it needs to strike a good balance between modularity and performance. On one hand, due to the complexity of modern layout engines, browser developers have to *manually* design scheduling strategies for rendering tree traversals in exchange for better performance. On the other hand, tree traversals in compilers are designed in a modular way, where mutually dependent traversals read and update attributes of ASTs [41]. This provides a great opportunity for *automated scheduling* of tree traversals. In particular, traversals that operate on the same node can be merged to reduce the overhead of node visiting and improve locality.

Even though manual scheduling of tree traversals offers fine-grained control for maximizing performance of a layout engine, the complexity of layout semantics (e.g., from W3C CSS standards) make it difficult to maintain the infrastructure and fix the notorious bugs. For instance, the Servo layout engine contains several bugs that have been open for over five years [14, 25] due to a mismatch between the intended semantics and the architecture chosen for its implementation[1].

Automated scheduling of tree traversals aims to merge modular passes (or visitors) that operate on the same node of a tree. However, existing approaches are far from satisfactory. For instance, there are approaches that are specialized to certain types of tree traversals,

[1]A Servo developer remarked that "it took three weeks before I realize[d] the actual complexity of the problem", which refers to the bug that is still open by the time of this submission; the Servo developers have resolved to delay fixing it until a complete rewrite of the layout engine is done [38].

such as TreeFuser [40] and Grafter [41]. But they leverage deterministic rewrite rules as well as automata-based representations that are complex to maintain. Synthesis-based tools like FTL [32] express tree computations using attribute grammars and leverage constraint solvers (i.e., Prolog) to find candidate solutions that satisfy the dependencies among tree computations. However, FTL requires domain experts for translating the layout semantics into constraints in Prolog, which is error-prone.

Motivated by these observations, we introduce HECATE, a synthesizer of tree traversals that can produce both serial and parallel traversals. In particular, HECATE provides a high-level tree language for defining *templates* for new kinds of traversals. Furthermore, the core synthesis engine of HECATE is built on top of a solver-aided framework [45], which lifts the execution of an interpreter for tree language programs into constraints that can be solved by off-the-shelf solvers. As a result, HECATE eliminates the enormous engineering efforts in FTL while preserving the efficiency and flexibility of exploring different design choices. To use HECATE to synthesize a concrete traversal, the developer only needs to specify a simple *traversal template* with holes yet to be filled with computation rules defined by the tree language. After that, HECATE completes the traversals using a counterexample-guided inductive synthesis (CEGIS) loop [43]: the synthesizer searches for a candidate traversal that works for the initial examples. The verifier then looks for a counterexample that fails for the traversal and invokes the synthesizer again to find a new candidate that is consistent with the counterexample. The process continues until the verifier cannot find additional counterexamples.

As we show later in the evaluation, direct interpretation of full semantics of a tree traversal will lead to difficult-to-solve constraints due to path explosions. To address this, HECATE employs a *domain-specific* symbolic compilation strategy, which maintains the usability of symbolic compilation, yet scales to problems orders of magnitude larger. The key insight is a *semantic projection* layer between the interpreter and the symbolic compilation engine that tailors the constraint generation procedure. Specifically, we introduce a *trace language* that disentangles complex dependencies from time domain to relational domain, where constraints can be equivalently expressed independent of time, thus clearing out path explosions while still ensuring the correctness of constraints. Under domain-specific symbolic compilation, the trace language generates integer linear programming (ILP) constraints that can be solved efficiently.

We implement HECATE and compare it against GRAFTER and FTL, showing that our tool is expressive, efficient, and flexible. On the set of GRAFTER problems, HECATE synthesizes traversals that trade off traversal fusion to exploit parallelism. Additionally, HECATE allows defining a tree data structure with an arbitrary number of children. Together, parallelism and data structure improvements accelerate the computation 2× on a tree rendering problem. Finally, HECATE's domain-specific symbolic compilation accelerates synthesis 3× compared to the general-purpose compilation to an SMT solver; when scheduling a CSS engine traversal, this ILP-based synthesis executes orders of magnitude faster.

To summarize, we make the following contributions:

- We propose a CEGIS framework for tree traversals.

```
class Box{
public:
  int w0,h0; // input (default)
  int w1,h1; // helper
  int w,h; // output (final)
}

class Inner: public Box{
public:
  Box* fc; // first child
  Box* nx; // next sibling
}
void Inner::calcWidth() {
  fc->calcWidth();
  nx->calcWidth();
  w = max( w0, fc->w1 );
  w1 = max( w, nx->w1 );
}
void Inner::calcHeight() {
  fc->calcHeight();
  nx->calcHeight();
  h = max( h0, fc->h1 );
  h1 = h + nx->h1;
}

class Leaf: public Box{
public:
  Box* nx; // next sibling
}
void Leaf::calcWidth() {
  nx->calcWidth();
  w = w0;
  w1 = max( w, nx->w1 );
}
void Leaf::calcHeight() {
  nx->calcHeight();
  h = h0;
  h1 = h + nx->h1;
}
```

(a) unfused version

```
/* same as unfused
*
*
*
*
*/

/* same as unfused
*
*
*
*/
void Inner::fusedCalc() {
  fc->fusedCalc();
  nx->fusedCalc();
  w = max( w0, fc->w1 );
  w1 = max( w, nx->w1 );
  h = max( h0, fc->h1 );
  h1 = h + nx->h1;
}






/* same as unfused
*
*
*/
void Inner::fusedCalc() {
  nx->fusedCalc();
  w = w0;
  w1 = max( w, nx->w1 );
  h = h0;
  h1 = h + nx->h1;
}
```

(b) fused version

Figure 1: Pseudo-code class definitions (unfused and fused versions) for rendering tree example.

- We propose a domain-specific trace language that disentangles complex dependencies from time domain to relational domain, which results in easy-to-solve constraints.
- We implement the proposed ideas in a tool called HECATE and demonstrate that it achieves 3× speed-up on GRAFTER benchmarks compared to general-purpose symbolic compilation.

2 OVERVIEW

In this section, we illustrate how HECATE works using a running example.

A Rendering Tree Example. Layout engines in modern web browsers utilize the box model in rendering procedures. This example demonstrates simplified behaviors of two types of boxes: `Inner` boxes and `Leaf` boxes where the former can hold child boxes and the latter can't. Figure 1(a) shows the realization of the boxes. In the example:

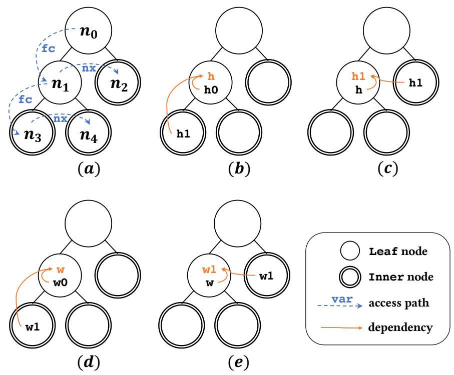


Figure 2: A motivating example tree with different properties shown (a): access paths; (b)-(e): dependencies between attributes of nodes.

- Line 16: The final width of an `Inner` box (denoted by `w`) is decided by the *larger* one between its default width (denote by `w0`) and the *maximum* width of its children (denoted by `fc->w1`).
- Line 22: The final height of an `Inner` box (denoted by `h`) is decided by the *larger* one between its default height (denoted by `h0`) and the *summed* height of its children (denoted by `fc->h1`).
- Line 33, 38: For a `Leaf` node, its final width and height are decided by its provided default values (denoted by `w0` and `h0`, respectively).
- Specifically, the helper variable `w1` records the maximum final width among a node and all its siblings accessible by `nx`, and the helper variable `h1` records the summed final height of a node and all its siblings accessible by `nx`. In other words, a node can refer to `w1` of its first child as the maximum final width among all its children, and refer to `h1` of its first child as the summed final height of all its children.

The two methods `calcWidth` and `calcHeight` demonstrate the computations for final width and height, respectively.

Typically, rendering a box requires a proper tree traversal to compute all the attribute values. Besides, in real-world use case scenarios, a finer-grained scheduling of computations is usually required as an optimization. As shown in Figure 1(b), a more efficient method `fusedCalc` is synthesized to perform width and height computations at the same time. Because attribute values may depend upon one another, solving for a correct order of attribute evaluations becomes challenging.

How* HECATE *Works. Now with HECATE, the user starts by providing: 1) a *symbolic traversal* as shown in Figure 4(a) with slots ι_i in which we can schedule *at most one* computation rule from the grammar, 2) an example tree shown in Figure 2(a), and 3) the corresponding semantics written in HECATE's visitor language (shown in Figure 3). HECATE then automatically synthesizes a concrete traversal by filling the slots with computation rules while respecting all the read-write dependencies, as shown in Figure 4(b).

More specifically, executing the traversal over the example tree on an `Inner` node (e.g., n_1) first recursively computes the attributes of its child nodes (line 3 in Figure 4(b), i.e., n_3 and n_4 in Figure 2(a)) and its next sibling (line 4, i.e., n_2), and then attributes of itself (line 5-8). For a `Leaf` node, the computation is similar but without the recursion on the children.

Figure 5 illustrates the overview of HECATE, which instantiates a CEGIS loop with two phases: 1) Given the specification that contains class definitions as attribute grammar, initial trees, and a symbolic traversal, the synthesizer searches for a concrete traversal, and sends it to the verifier; 2) The verifier checks the correctness of the concrete traversal over all possible example trees (up to depth k)

```
interface Box{
  input w0,h0: int;
  output w1,w,h1,h: int;
}
class Inner: Box{
  children {
    nx : Optional[Box];
    fc : Optional[Box];
  }
  rules {
    self.w := max( self.w0, fc.w1 );
    self.w1 := max( self.w, nx.w1 );
    self.h := max( self.h0, fc.h1 );
    self.h1 := self.h + nx.h1;
  }
}
class Leaf: Box{
  children {
    nx : Optional[Box];
  }
  rules {
    self.w := self.w0;
    self.w1 := max( self.w, nx.w1 );
    self.h := self.h0;
    self.h1 := self.h + nx.h1;
  }
}
```

Figure 3: Class definitions in HECATE for rendering tree example.

```
1  traversal layout {
2    case Inner{
3      recur fc;
4      recur nx;
5      ι0;
6      ι1;
7      ι2;
8      ι3;
9    }
10   case Leaf{
11     recur nx;
12     ι4;
13     ι5;
14     ι6;
15     ι7
16   }
17 }
```

(a) symbolic

```
1  traversal layout {
2    case Inner{
3      recur fc;
4      recur nx;
5      eval self.w;
6      eval self.h;
7      eval self.w1;
8      eval self.h1;
9    }
10   case Leaf{
11     recur nx;
12     eval self.w;
13     eval self.h;
14     eval self.w1;
15     eval self.h1;
16   }
17 }
```

(b) concrete

Figure 4: Symbolic and concrete tree traversals for rendering tree example.

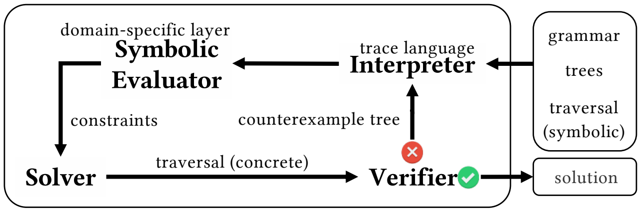


Figure 5: Overview of Hecate.

and returns a counterexample tree that fails the current traversal. Then the synthesizer finds a new candidate that is consistent with the newly added counterexample tree. This process continues until the verifier can not find additional counterexamples to refute the current traversal, which will be returned as the correct solution.

To synthesize the desired traversal, Hecate leverages Rosette [45] to lift an interpreter for tree traversal into a synthesizer through *symbolic compilation*. In particular, given a *symbolic traversal* with slots, symbolic compilation expands each slot into all possible choices of computation rules, and executes the traversal to generate dependency constraints under different choices. A concrete traversal is then obtained by solving the constraints using off-the-shelf SMT solvers [18, 24].

While a faithful interpretation of the semantics of the traversal usually causes path explosions, as choices of rules depend on choices made on preceding execution steps, it generates complex SMT constraints that are hard to solve, thus damaging the performance. Our solution is to insert a domain-specific layer below the interpreter and above the symbolic engine (i.e., Rosette), which exposes a domain-specific *trace language* that still allows us to write the interpreter conveniently, while avoiding explicit path conditions and generating compact ILP constraints that are easier to solve by off-the-shelf ILP solvers [24].

3 PROBLEM FORMULATION

In order to precisely describe our synthesis problem, we first present some definitions that we use throughout the paper.

3.1 Attribute Grammar for Tree Visitors

To represent tree structures and their visitors that contain a set of computation rules over attributes, we introduce a tree visitor language $\mathcal{L}_a$ based on *attribute grammar* [26]. Figure 6 defines the syntax of $\mathcal{L}_a$ in an object-oriented style:

- A class (i.e., ⟨*class*⟩) represents the type(s) of a tree node.
- A node can refer to its child nodes via ⟨*children*⟩ block, and each node also stores values of primitive fields — we call these fields *attributes*.
- A ⟨*rules*⟩ block contains *computation rules* and each rule is a ⟨*cstmt*⟩ statement.
- The left-hand-side (LHS) of a statement specifies an *access path* ⟨*sel*⟩ to an attribute that is computed by the expression in the right-hand-side (RHS).
- An expression on the RHS can be: constants, binary expressions, branches, aggregations, and function calls, etc.
- Each attribute can be assigned exactly once.

⟨*interface*⟩	::=	interface ⟨*id*⟩ { (⟨*tup*⟩;)* }
⟨*class*⟩	::=	class ⟨*tup*⟩ { ⟨*children*⟩ ⟨*rules*⟩ }
⟨*children*⟩	::=	children { (⟨*tup*⟩;)* }
⟨*rules*⟩	::=	rules { (⟨*cstmt*⟩;)* }
⟨*tup*⟩	::=	⟨*id*⟩:⟨*id*⟩(,⟨*id*⟩)*
⟨*sel*⟩	::=	⟨*id*⟩(.⟨*id*⟩)?.⟨*id*⟩
⟨*expr*⟩	::=	⟨*const*⟩ | ⟨*sel*⟩ | *f*(⟨*expr*⟩*)
	|	⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ | fold(⟨*expr*⟩+)
	|	if ⟨*expr*⟩ then ⟨*expr*⟩ else ⟨*expr*⟩
⟨*cstmt*⟩	::=	⟨*sel*⟩ := ⟨*expr*⟩
⟨*op*⟩	::=	+ | − | × | ÷ | ...

f ∈ **functions** ⟨*const*⟩ ∈ **constants** ⟨*id*⟩ ∈ **identifiers**

Figure 6: Syntax for attribute grammar $\mathcal{L}_a$.

⟨*traversal*⟩	::=	traversal ⟨*id*⟩ { ⟨*case*⟩* }
⟨*case*⟩	::=	case ⟨*id*⟩ { (⟨*tstmt*⟩;)* }
⟨*recur*⟩	::=	recur ⟨*node*⟩
⟨*iterate*⟩	::=	iterate { (⟨*tstmt*⟩;)* }
⟨*parallel*⟩	::=	parallel { (⟨*tstmt*⟩;)* }
⟨*eval*⟩	::=	eval ⟨*cstmt*⟩
⟨*tstmt*⟩	::=	ι | ⟨*recur*⟩ | ⟨*iterate*⟩ | ⟨*eval*⟩

⟨*id*⟩ ∈ **identifiers** ⟨*node*⟩ ∈ **nodes**

Figure 7: Syntax for tree traversal language $\mathcal{L}_t$.

As a result, $\mathcal{L}_a$ is specialized for modeling the behaviors of reading and writing of attributes of the current node and its children, which essentially describes the dependency relations between attributes of nodes.

Example 3.1. The code snippet from Figure 3 declares the attribute grammar for nodes of types `Inner` and `Leaf`, which share the same set of attributes declared by the interface `Box`. Specifically:

- each `Inner` node has two children `nx` and `fc` that point to its next sibling and first child respectively;
- a `Leaf` node does not have children.

Rules for computing the attributes vary across different classes. e.g.,

- attributes `w` and `h` of a `Leaf` node only depend on attributes from itself,
- attribute `w` from an `Inner` node depends on the default width of itself (i.e., `self.w0`) and the maximum width of its children (i.e., `fc.w1`),
- attribute `h` from an `Inner` node depends on the default height of itself (i.e., `self.h0`) and the summed height of its children (i.e., `fc.h1`).

3.2 Language for Tree Traversals

To formally define a tree traversal, we first introduce domains for different notations.

Syntax. Figure 7 summarizes the language $\mathcal{L}_t$ for expressing tree traversals. In particular, A traversal ⟨*traversal*⟩ is declared with a list of ⟨*case*⟩ blocks. Each ⟨*case*⟩ block matches a node type and contains statements of the following forms:

- $\langle recur \rangle$ *recursively* visits a child of the current node,
- $\langle iterate \rangle$ *iteratively* visits every child of the current node,
- $\langle parallel \rangle$ *in parallel* visits every child of the current node[2],
- $\langle eval \rangle$ evaluates an attribute computation rule, and
- a slot ι represents zero or one computation rule *yet to be determined*[3].

Domains. Our system contains the following domains:

- Each traversal operates over a set of trees denoted by $\mathbb{E}$.
- Each tree contains a set of nodes (i.e., $\mathbb{N}$).
- A time domain $t \in \mathbb{T}$ enforces the order of different rules.
- An attribute grammar provides a list of attributes $\mathbb{A}$ that uniquely determine their corresponding computation rules.
- Each traversal provides a list of *slots* (i.e., $\mathbb{I}$) for holding computation rules from the attribute grammar.
- Each node n has a set of locations (i.e., $\mathbb{L}$) that refer to its corresponding attributes in runtime.

Definition 3.2. **Traversal**. Given a tree, a *traversal* defines a total order relation $\prec$ over the set of all locations of the tree.

Example 3.3. A concrete *post-order* traversal (i.e., Figure 4(b)) on the tree in Figure 2 yields the following total order of locations:

$$
\begin{aligned}
&n_4.\mathtt{w} \prec n_4.\mathtt{h} \prec n_4.\mathtt{w1} \prec n_4.\mathtt{h1} \prec n_3.\mathtt{w} \prec n_3.\mathtt{h} \prec n_3.\mathtt{w1} \prec n_3.\mathtt{h1}\\
\prec &n_1.\mathtt{w} \prec n_1.\mathtt{h} \prec n_1.\mathtt{w1} \prec n_1.\mathtt{h1} \prec n_2.\mathtt{w} \prec n_2.\mathtt{h} \prec n_2.\mathtt{w1} \prec n_2.\mathtt{h1}\\
\prec &n_0.\mathtt{w} \prec n_0.\mathtt{h} \prec n_0.\mathtt{w1} \prec n_0.\mathtt{h1}
\end{aligned}
$$

where in every *time step* $t \in \mathbb{T}$ one location is visited. Note that different traversals may induce different orders.

A traversal is *symbolic* if it contains at least one slot ι and is *concrete* otherwise.

Example 3.4. Figure 4(a) declares a symbolic and post-order traversal over nodes of types `Inner` and `Leaf`. Figure 4(b) is an instantiation of the previous symbolic traversal. In particular, the traversal first computes the attributes for the leaves of type `Leaf`, and then the attributes of the nodes of type `Inner`.

Note that the concrete post-order traversal preserves the *read-write dependencies* induced by the attribute grammar in Figure 3. On the contrary, a pre-order traversal would not be *valid* since it violates the read-write dependencies imposed by attributes including `w` and `h` of a `Inner` node. We then define the *tree traversal synthesis* problem as follows:

Definition 3.5. **Tree Traversal Synthesis**. Given an attribute grammar $\mathcal{L}_a$ and a symbolic traversal $\hat{P}_t$ with holes, a tree traversal synthesis problem is to induce a concrete traversal P_t by completing the holes ι in $\hat{P}_t$ with computation rules in $\mathcal{L}_a$, such that for arbitrary instantiated trees from $\mathcal{L}_a$: 1) all attributes are computed, and 2) all read-write dependencies are preserved.

Example 3.6. Given the attribute grammar $\mathcal{L}_a$ in Figure 3 and the symbolic traversal in Figure 4(a), Hecate synthesizes the concrete traversal in Figure 4(b). Given arbitrary tree derived from Figure 2, the synthesized traversal computes all attributes of the tree *exactly once* and respects all read-write dependencies.

[2]If parallelism is impossible, Hecate falls back to $\langle iterate \rangle$.
[3]One can extends the semantics of ι to represent one or *more* rules.

As we show later, to bypass the challenge of complex SMT constraints that are generated by a faithful interpretation of a traversal's semantics, a scalable approach to solve the placement and resource allocation problems is to use ILP to map the computation rules to the available slots in the traversal [15, 36].

Definition 3.7. **0-1 Integer Linear Programming**. Given coefficients a, b and c, the 0-1 ILP problem is to solve for x as follows:

$$\min \sum_i c_i x_i \quad s.t.\ \ \forall a_{i,j}.\ \sum_j a_{i,j} x_j \leq b_i,$$

where all entries are integers and in particular $x_j \in \{0, 1\}$.

We obtain a set of ILP constraints that is easy to solve by ILP solvers from domain-specific symbolic compilation via program written in trace language, which is deferred to Section 5 for a detailed discussion.

4 TREE TRAVERSAL SYNTHESIS

In this section, we formally introduce our synthesis framework for tree traversals that is based on counterexample-guided inductive synthesis (CEGIS). In what follows, we first give a high-level overview of the synthesis framework, then we show how to reduce the synthesis problem to a general-purpose symbolic compilation problem based on Rosette. Finally, we briefly discuss its limitation.

4.1 System Overview

As shown in Figure 5, Hecate takes as inputs an attribute grammar $\mathcal{L}_a$, a symbolic traversal with unknown slots in $\mathcal{L}_t$, and an initial tree for validating the correctness of the traversal. The output of Hecate is a concrete traversal that respects all the read-write dependencies imposed by the attribute grammar.

Synthesis. Figure 8(a) sketches the core synthesis engine that is built on top of Rosette [45], a hybrid symbolic compiler that combines symbolic execution and bounded model checking to compute compact constraints. In particular, the general-purpose interpreter `interpret` for tree traversals takes as inputs a grammar `grammar`, a traversal `traversal`, and a concrete tree `tree`. Following the total order induced by `traversal`, the outermost loop of the interpreter recursively visits each node in `tree` and its corresponding locations `loc` (line 2). When evaluating a symbolic choice for a slot, symbolic evaluation considers each alternative concrete rule (line 3-4), generates the constraints stating that the dependencies are ready and the target has not been computed (line 6-7), sets the target attribute as ready, and updates the program state (line 8). The interpreter behaves as a regular emulator when it runs with concrete traversals and trees. For instance, running it with the post-order traversal in Figure 4(b) and the example tree in Figure 2 will pass all assertions and terminate normally. What is more interesting is that given a *symbolic traversal* `traversal` with slots yet to be filled, Rosette runs the interpreter with `traversal` and a concrete tree `tree` under symbolic evaluation; this encodes all possible concrete traversals that preserve the dependencies in `tree`, effectively lifting the interpreter to be a *synthesizer.*

```
1 (define (interpret grammar traversal tree)
2   (for ([loc (traversal grammar tree)])
3     (let* ([rule (get-rule loc)]
4            [node-attr (get-lhs rule loc)])
5       (for ([dep-attr (get-rhs rule loc)])
6         (assert (ready? dep-attr)))
7       (assert (not (ready? node-attr)))
8       (set-ready! node-attr))))
```

(a) general-purpose interpreter

```
(define (interpret grammar traversal tree)
  (for ([loc (traversal grammar tree)])
    (let* ([rule (get-rule loc)]
           [node-attr (get-lhs rule loc)])
      (for ([dep-attr (get-rhs rule loc)])
        (read dep-attr))
      (write node-attr))))
```

(b) domain-specific interpreter

Figure 8: Code snippets of general-purpose interpreter and domain-specific interpreter.

Verification. To ensure that the synthesized traversal `traversal` is not only correct on the initial example `tree` but also on all possible trees [4], we again leverage Rosette to build our verifier. In particular, the core of the verifier is another interpreter that is almost identical to the one in Figure 8(a). Now the inputs of the interpreter include a concrete traversal `traversal` that needs to be verified, as well as a *symbolic tree* $\hat{tr}$ that encodes the space of all possible concrete examples up to depth k. In that case, symbolically evaluating $\hat{tr}$ on `traversal` yields a formula stating that `tree` is correct on all instantiations of $\hat{tr}$. If the formula is satisfiable, the verifier then returns a counterexample to the synthesizer that will look for another candidate. Similar to prior work in Neo [20] and Bonsai [15], our symbolic tree $\hat{tr}$ is encoded as a bounded m-ary tree derived from the attribute grammar. We omit the details since it is not the main contribution.

We call the interpreter in Figure 8(a) *general-purpose symbolic compilation*. By nature, its encoding (general-purpose encoding) establishes read-write dependencies across different execution time steps. While general-purpose encoding is fairly intuitive and straightforward to implement, it may lead to complex constraints that are difficult to solve.

4.2 General-Purpose Symbolic Compilation

In this section, we elaborate on the details behind general-purpose symbolic compilation. Using the domains introduced in Section 3.2, we first define a set of *relational operators* to formalize the general-purpose symbolic compilation:

- *Assignment.* The assignment operator maps an attribute $a \in \mathbb{A}$ and a slot $\iota \in \mathbb{I}$ to a boolean variable in $\mathbb{B}$, i.e., $\sigma : \mathbb{A}\times\mathbb{I} \to \mathbb{B}$. Since each attribute a is uniquely computed by a computation rule from the attribute grammar, predicate $\sigma(a, \iota)$ evaluates to `true` iff the rule for computing attribute a is scheduled at slot ι.
- *Dereference.* Recall that in Figure 6, each computation rule for an attribute is composed of multiple *access paths* sel appearing in a statement. During tree traversal, the dereference operator $\zeta(n, \text{sel})$ returns the concrete location l (i.e., $\mathbb{N} \times \mathbb{A}$) that access path sel of node n points to.
- *Ready Bit.* The ready bit operator maps a node $n \in \mathbb{N}$, an attribute $a \in \mathbb{A}$, and a time step $t \in \mathbb{T}$ to a boolean variable in $\mathbb{B}$, i.e., $\delta : \mathbb{N}\times\mathbb{A}\times\mathbb{T} \to \mathbb{B}$. Here, predicate $\delta(n, a, t)$ returns `true` iff the attribute a of node n is already computed (i.e., ready for being read by other computation rules) before time step t.
- *Symbolic Choice.* Recall that a symbolic traversal $\hat{P}_t$ contains at least one slot ι that represents at most one attribute computation yet to be scheduled. To handle this case, we introduce a symbolic choice operator `choose` for non-deterministically choosing an attribute (to compute) from a list of available attributes $\mathbb{A}$. For instance, `(choose [Inner.h, Inner.w, none])` returns one of the attributes from the list.

Now, during the general-purpose symbolic compilation in Figure 8(a), the interpreter executes statements in traversal `trav` and dynamically inserts assertions (line 6-7) to state the correctness of every single computation rule. In particular, the correctness enforces *read-write dependency* using the *ready bit* operator δ. Therefore, for a statement with chosen attribute, e.g., `eval self.h` at slot ι_2 of node n_1, Rosette compiles it into the following constraints:

$$\delta(\zeta(n_1, \texttt{self.h0}), t) \wedge \delta(\zeta(n_1, \texttt{fc.h1}), t) \wedge \neg\delta(\zeta(n_1, \texttt{self.h}), t)$$

where t is the current time step and rule `self.h := max( self.h0, fc.h1 )` (line 13 in Figure 3) is used to compute attribute `self.h` for node n_1 of type `Inner`. Here, the above constraints state two properties about the read-write dependencies: 1) attributes of nodes (i.e., `self.h0` for n_1 and `fc.h1` for n_3) should be ready before they are read, and 2) the attribute of a node (i.e., `self.h` for n_1) should *not* be ready until it is written.

The interpreter starts by executing a *slot statement* ι in the symbolic traversal. In that case, each ι is dynamically replaced by a statement that non-deterministically chooses an available attribute a_i to schedule [5]: `eval (choose [`$\mathsf{a}_1, ..., \mathsf{a}_n$`])`. After that, Rosette symbolically evaluates the above statement and compiles it into a formula stating all possible cases where each case is *guarded* by the conjunction of assignment operators σ that represent the cumulative choices so far.

Example 4.1. For instance, at time step t, when the interpreter executes slot ι_2, i.e. line 7 in Figure 4(a), on `Inner` node n_1 in Figure 10(a), it can choose one of the five options from `none`, `Inner.w1`, `Inner.w`, `Inner.h1` and `Inner.h` according to its attribute grammar in Figure 10(b):

`eval (choose S)`

where

`S : [none, Inner.w1, Inner.w, Inner.h1, Inner.h]`

[4] Similar to prior work [15], we verify trees up to depth k.

[5] We omit the empty case for simplicity.

which is further transformed into the following formula:

$$
\begin{aligned}
&(\sigma(\mathtt{none}, \iota_2) \implies \mathit{true})\\
\vee\ &(\sigma(\mathtt{Inner.w1}, \iota_2) \implies \delta(\zeta(n_1, \mathtt{self.w}), t) \wedge \delta(\zeta(n_1, \mathtt{nx.w1}), t)\\
&\qquad \wedge \neg\delta(\zeta(n_1, \mathtt{self.w1}), t))\\
\vee\ &(\sigma(\mathtt{Inner.w}, \iota_2) \implies \delta(\zeta(n_1, \mathtt{self.w0}), t) \wedge \delta(\zeta(n_1, \mathtt{fc.w1}), t)\\
&\qquad \wedge \neg\delta(\zeta(n_1, \mathtt{self.w}), t))\\
\vee\ &(\sigma(\mathtt{Inner.h1}, \iota_2) \implies \delta(\zeta(n_1, \mathtt{self.h}), t) \wedge \delta(\zeta(n_1, \mathtt{nx.h1}), t)\\
&\qquad \wedge \neg\delta(\zeta(n_1, \mathtt{self.h1}), t))\\
\vee\ &(\sigma(\mathtt{Inner.h}, \iota_2) \implies \delta(\zeta(n_1, \mathtt{self.h0}), t) \wedge \delta(\zeta(n_1, \mathtt{fc.h1}), t)\\
&\qquad \wedge \neg\delta(\zeta(n_1, \mathtt{self.h}), t))
\end{aligned}
$$

where $\sigma(\mathtt{Inner.h}, \iota_2)$ evaluates to `true` iff we decide to compute attribute `Inner.h` at slot ι_2 using its corresponding rule `self.h := max( self.h0, fc.h1 )` from class `Inner`. Here in particular, every assignment predicate σ implies a conjunction of three ready bit predicates δ asserting corresponding properties of read-write dependencies. For example, the last clause

$$
\begin{aligned}
\sigma(\mathtt{Inner.h}, \iota_2) \implies &\ \delta(\zeta(n_1, \mathtt{self.h0}), t) \wedge \delta(\zeta(n_1, \mathtt{fc.h1}), t)\\
&\wedge \neg\delta(\zeta(n_1, \mathtt{self.h}), t)
\end{aligned}
$$

indicates that in order to schedule rule `Inner.h` at slot ι_2,

- `self.h0` and `fc.h1` should be ready before time step t, and
- `self.h` should *not* be scheduled before time step t.

In addition to correctness constraints, we also enforce auxiliary constraints to induce valid traversals. For instance, the following constraint requires every slot be filled with *at most one* rule:

$$\forall \iota. (\bigvee_{a_0} \bigwedge_{a \neq a_0} \neg\sigma(a, \iota) \wedge \sigma(a_0, \iota)) \vee (\bigwedge_{a} \neg\sigma(a, \iota)).$$

And the following requires every rule be used by only one slot:

$$\forall a. \bigvee_{\iota_0} \bigwedge_{\iota \neq \iota_0} \neg\sigma(a, \iota) \wedge \sigma(a, \iota_0).$$

Performance Analysis. While it is intuitive and straightforward to build a tree traversal synthesizer using general-purpose encoding, it suffers from path explosion by faithfully following the execution of a traversal, even with the effective state-merging and pruning strategy from Rosette. Figure 9 shows how the number of symbolic state grows as time goes by. Consider a tree of n nodes with an average of k slots per node, the general-purpose symbolic compilation will generate constraints based on a chain of length $n \cdot k$ with dependencies between choices made in a recursive way, i.e. nested choose operations. Assuming that every slot has a candidate set of q rules to fill in on average, the total number of symbolic states after compilation can be *up to* $q^{n \cdot k}$. As shown in our evaluations, the general-purpose symbolic compilation creates constraints that take a long time to solve.

5 DOMAIN-SPECIFIC SYMBOLIC COMPILATION

As discussed in Section 4, a general-purpose symbolic compilation faithfully follows the execution of a traversal across different execution time steps, which leads to constraints that are hard to solve. To mitigate this problem, we propose a domain-specific trace language, which projects the complex dependencies from *time domain* to *relational domain* and yields easy-to-solve constraints. In what follows, we first introduce the trace language $\mathcal{L}_r$, and then

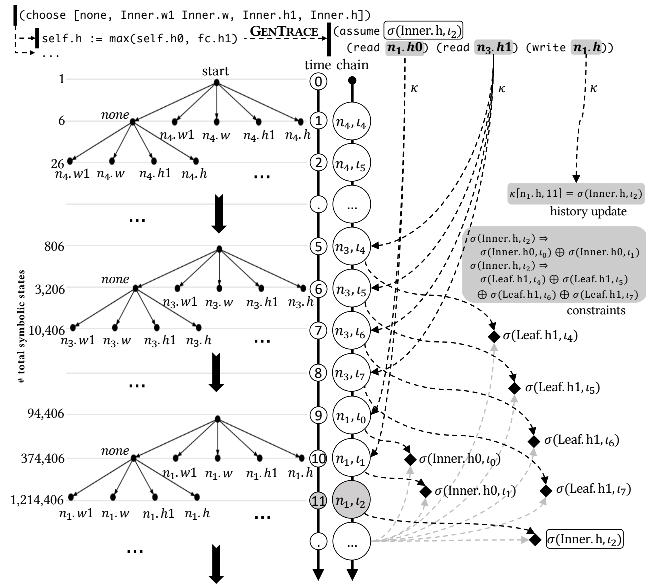
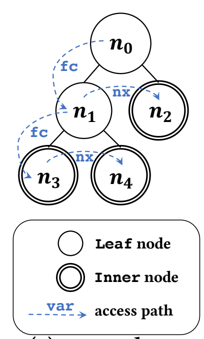


Figure 9: General-purpose (left) v.s. domain-specific (right) symbolic compilations.




```
class Inner: Box{
  children {
    nx : Optional[Box];
    fc : Optional[Box];
  }
  rules {
    self.w := max( self.w0, fc.w1 );
    self.w1 := max( self.w, nx.w1 );
    self.h := max( self.h0, fc.h1 );
    self.h1 := self.h + nx.h1;
  }
}
```

(a) example tree **(b) visitor program snippet**

Figure 10: The motivating example tree as in Section 2 and its corresponding visitor program snippet.

show how to obtain ILP constraints via domain-specific symbolic compilation.

5.1 A Trace Language for Tree Traversals

As shown in Figure 8(b), thanks to Rosette, the skeleton of domain-specific interpreter for synthesizing tree traversals can be obtained with a minor modification over the general-purpose version: upon executing a statement in traversal `traversal`, instead of directly adding its corresponding assertions, we first translate the statement into another program in *trace language* $\mathcal{L}_r$, and then leverage Rosette to lift the execution of the new trace program to constraints that can be modeled as an *integer linear programming* (ILP) problem.

Table 1: Operations of the symbolic trace language $\mathcal{L}_r$.

Operation	Description
`(choose [`$a_1, ... a_n$`])`	choose one from the attributes
`(alloc)`	returns a fresh concrete location
`(read` $n.a$`)`	logs a read from $n.a$
`(write` $n.a$`)`	logs a write to $n.a$

The syntax and semantics of $\mathcal{L}_r$ are summarized in Table 1. Intuitively, $\mathcal{L}_r$ understands dependency relations carried through attributes on nodes with fully abstract contents. In particular:

- `(read` $n.a$`)` logs the read action of attribute a on node. n;
- `(write` $n.a$`)` logs the write action of attribute a on node n.
- `(choose [`$a_1, ..., a_n$`])` non-deterministically selects an attribute a_i to compute.

For the sake of simplicity, we use the built-in `assume` function in Rosette to explicitly enumerate each option of symbolic choices under different assumptions.

During the execution of the domain-specific interpreter (Figure 8(b)) at line 6, Hecate invokes a syntax-directed transpilation procedure to generate the corresponding trace program, which captures the dependency relations to ensure the correctness of tree traversals, and provides succinct statements that eventually lead to efficient constraints (Section 5.2).

Example 5.1. Following Example 4.1, suppose we are in the symbolic traversal (i.e., Figure 4(a)) at slot ι_3 and the current node is n_0 in Figure 10(a). And the synthesizer decides to select a rule `self.h := max( self.h0, fc.h1 )` from the visitor program in Figure 10(b) to compute the `Inner.h` attribute in slot ι_3, then a syntax-directed transpilation procedure is invoked to generate the following trace program:

$$(\texttt{assume}\ \sigma(\texttt{Inner.h}, \iota_3)\quad (\texttt{read}\ n_0.\texttt{h0})\ (\texttt{read}\ n_1.\texttt{h1})\ (\texttt{write}\ n_0.\texttt{h})).$$

Semantically the above trace program states that in order to compute `Inner.h` at slot ι_3, two attributes (i.e., n_0.`h0` and n_1.`h1`) should first be read and another attribute (i.e., n_0.`h`) should then be written. The trace program records read-write dependencies in a more compact way without introducing time steps.

5.2 Symbolic Compilation of Trace Program

Even if we obtain a trace program P_r using the procedure discussed in Section 5.1, the trace program itself does not mitigate the path explosion problem because similar to the general-purpose encoding, the symbolic choice statements in the trace program still encode path conditions at each time step. To address this challenge, we discuss how our domain-specific compilation further *projects* the executions of the trace program into *compact* constraints that can be solved by efficient ILP solvers [24].

Dependency Constraints. We first introduce a *dependency* operator κ that takes as inputs a location $l \in \mathbb{N} \times \mathbb{A}$, a time step $t \in \mathbb{T}$, and returns a boolean variable σ that specifies all possible slots in which the attribute a of location l was computed. In other words, κ encodes the dependency between locations (i.e., `read`) and their corresponding attributes (i.e., `write`).

We then illustrate the relationship between the trace program and the dependency operator κ. In particular, for a `write` instruction `(write` $n.a$`)` at time step t guarded by $\sigma(a, \iota)$, the dependency operator κ gets updated by:

$$\kappa(n.a, t) \leftarrow \sigma(a, \iota),$$

where attribute a is written at time step t if $\sigma(a, \iota)$ evaluates to 1 (i.e., `true`).

For a `read` instruction `(read` $n.a$`)` at time step t guarded by $\sigma(a, \iota)$, if $\sigma(a, \iota)$ evaluates to `true`, then it *implies* that attribute a must be written somewhere *before* time step t. Formally speaking, we have constraint:

$$\sigma(a, \iota) \implies (\exists\, t_0.(t_0 < t) \wedge \kappa(n.a, t_0)),$$

which can be easily translated into its equivalent ILP constraint [6]:

$$\sigma(a, \iota) \leq \sum_{t_0 < t} \kappa[n.a, t_0], \qquad \text{(read constraint)}$$

Example 5.2. Following Example 4.1 but in domain-specific symbolic compilation, as shown in Figure 9, suppose we want to schedule `self.h := max( self.h0, fc.h1 )` at slot ι_2 of node n_1, which corresponds to the following trace program:

$$(\texttt{assume}\ \sigma(\texttt{Inner.h}, \iota_2)\quad (\texttt{read}\ n_1.\texttt{h0})\ (\texttt{read}\ n_3.\texttt{h1})\ (\texttt{write}\ n_1.\texttt{h})).$$

The domain-specific encoding compiles the above trace program into the following ILP constraints:

$$\begin{aligned}
\sigma(\texttt{Inner.h}, \iota_2) &\leq \sum_{t_0 < t} \kappa[n_1.\texttt{h0}, t_0] \\
&= \sigma(\texttt{Inner.h0}, \iota_0) + \sigma(\texttt{Inner.h0}, \iota_1), \qquad \text{(read for } n_1.\texttt{h0})\\
\sigma(\texttt{Inner.h}, \iota_2) &\leq \sum_{t_0 < t} \kappa[n_3.\texttt{h1}, t_0] \\
&= \sigma(\texttt{Leaf.h1}, \iota_4) + \sigma(\texttt{Leaf.h1}, \iota_5) \\
&\quad + \sigma(\texttt{Leaf.h1}, \iota_6) + \sigma(\texttt{Leaf.h1}, \iota_7), \qquad \text{(read for } n_3.\texttt{h1})
\end{aligned}$$

where t corresponds to the time step when visiting ι_2 of node n_1. According to Definition 3.2, since a traversal defines a total order relation over all locations of a tree, we can map the location that is currently being evaluated to a certain time step t, and generate constraints that require all the dependencies of this location are ready before time step t. This is done by κ in the example. Then, we again utilize the mapping to cancel the time step variables in the constraints by mapping them back to potential locations, thus resulting in a more compact constraint system.

Validity Constraints. Similar to the general-purpose encoding in Section 4.2, we also impose extra constraints to ensure the validity of the traversals.

- For every slot ι, at most one rule can be filled in:

$$\forall \iota. \sum_{a} \sigma(a, \iota) \leq 1, \qquad \text{(slot constraint)}$$

- Every rule a is used by exact one slot ι:

$$\forall a. \sum_{\iota} \sigma(a, \iota) = 1. \qquad \text{(rule constraint)}$$

[6] Interchangeably, we use the same domain notation B to denote the boolean domain and 0-1 integer domain for general-purpose and domain-specific symbolic encodings, respectively.

Performance Analysis. To understand why the domain-specific compilation generates better constraints than the general-purpose version, we use Figure 9 to show a comparison between two strategies. Here we use the total number of symbolic states (i.e., the input space of all relational operators that introduce symbolic states) to *approximate* the complexity of constraints. Both the assignment operator $\sigma : \mathbb{A} \times \mathbb{I} \rightarrow \mathbb{B}$ and the readiness operator $\delta : \mathbb{N} \times \mathbb{A} \times \mathbb{T} \rightarrow \mathbb{B}$ introduce symbolic states. Even though the size of the symbolic states generated by the readiness operator can grow as the size of the tree n, the number of attributes q, and the number of slots k increase, it's still bounded by a polynomial growth. In particular, the domain-specific encoding generates a maximum of $(1+n^2) \cdot q \cdot k$ symbolic states, which is more compact and less complex than the exponential number generated by general-purpose encoding.

6 EVALUATION

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

(1) (Expressiveness) Is Hecate's tree (visitor/traversal/trace) language expressive enough? In particular, can it express prevailing tree traversal synthesis problems and solve them?
(2) (Performance) What is the performance of synthesized traversals, compared to those generated by state-of-the-art traversal synthesizers?
(3) (Flexibility) Can Hecate be extended to explore traversals of different design choices?
(4) (Efficiency) What is the benefit of the domain-specific encoding compared to general-purpose encoding?

For all experiments, Hecate requires user-provided attribute grammar, a symbolic traversal and an initial example tree as input, and outputs a concrete traversal in tree traversal language $\mathcal{L}_t$.

6.1 Comparison against Grafter

We first compare Hecate against Grafter [41], the state-of-the-art tree traversal synthesizer based on static dependence analysis. In particular, Grafter builds *access automata* that summarises dependency relations for tree visitors, and synthesizes tree traversals using a deterministic algorithm. We adapt the original benchmark set from Grafter, which contains five representative tree traversal synthesis problems from real-world applications. Since Grafter benchmarks are written in C++, we also implement a code generator for converting concrete traversals synthesized by Hecate into corresponding C++ versions through syntax-directed translation. To study the benefit of domain-specific encoding discussed in Section 5, we also implement general-purpose encoding discussed in Section 4.2, which we denote as Hecate$^{\mathbb{G}}$.

Efficiency and Expressiveness. Table 2 shows the results of the comparison. In particular, Hecate supports all 5 benchmarks from Grafter and successfully synthesizes the correct solutions (i.e., traversals that are semantically equivalent to the ones generated by Grafter.) within 5.9 seconds on average. Specifically, Hecate yields an averaged speed-up of 3.1× compared to Hecate$^{\mathbb{G}}$ and 8.0× compared to Grafter. The evaluation shows that Hecate's tree language is expressive to support a variety of tree traversal applications. Furthermore. the comparison between Hecate and Hecate$^{\mathbb{G}}$ also demonstrates the benefits of domain-specific encoding.

Table 2: Comparison between Grafter, Hecate and Hecate$^{\mathbb{G}}$ (with general-purpose encoding). The table shows total synthesis time (synthesis + verification) in second.

Benchmark	# of Rules	Grafter	Hecate	Hecate$^{\mathbb{G}}$
BinaryTree	16	2.6	1.1	3.2
FMM	14	7.6	1.0	1.6
Piecewise	12	12.6	2.1	3.1
AST	136	151.7	20.6	73.4
RenderTree	50	62.0	4.1	10.1

Performance. To evaluate the performance of the synthesized traversals, we directly adopt the workload from Grafter. Since our symbolic traversals are written in a way to "fuse" tree visitors whenever possible, like Grafter, the performance of our synthesized traversals are almost identical to the ones generated by Grafter. However, unlike Grafter that uses a deterministic algorithm for generating one unique solution for each benchmark, the tree language enables Hecate to flexibly explore various traversals of different design choices, some of which lead to dramatic performance speed-up. In what follows, we elaborate on the details using a case study from one of Grafter's benchmarks: RenderTree. [7]

Usability. To further minimize user effort, we implement a variant Hecate$^{\mathbb{A}}$ that incorporates an auto-tuner that can *automatically* search for useful symbolic traversals during synthesis. In particular, the user only has to provide attribute grammar, and Hecate$^{\mathbb{A}}$ will construct the example trees and initiate an outer loop that searches for a symbolic traversal that ensures correctness of its corresponding synthesized concrete traversal. Our experimental results indicate Hecate$^{\mathbb{A}}$ can solve four Grafter benchmarks as fast as Hecate; for the AST benchmark with complex symbolic traversals, it takes Hecate$^{\mathbb{A}}$ more than 30mins to find a solution. We show that it is possible to get rid of more manual inputs for Hecate using a simple auto-tuner.

6.2 Case Study: RenderTree

In the RenderTree benchmark, a document tree consists of a list of pages containing nested horizontal and vertical containers with concrete elements as leaf nodes (e.g., text boxes, images, and itemized lists). A total of five rendering passes compute various visual attributes: (1) resolving flexible widths, (2) resolving relative widths, (3) computing heights, (4) propagating font styles, and (5) finalizing positions of elements. Each pass potentially depends on the attributes computed by previous passes.

Unlike Grafter, which only generates one unique traversal that fuses tree visitors whenever possible, Hecate offers a number of design choices. For instance, tree nodes frequently visit their children, which can be modeled using either linked lists or vectors. Moreover, when the children have no dependencies between themselves, the user may parallelize the computations using the

[7] See Appendix A for a detailed case study for another benchmark: AST.

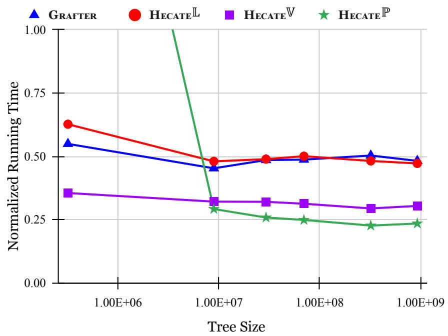


Figure 11: Running time of fused traversals compared to the unfused baseline.

```
class Inner: Box{
  children {
    cs: [Box]; // vector of 0 or more elements
  }
  rules {
    self.w := fold( max, self.w0, cs.w );
    self.h1 := fold( +, 0, cs.h );
    self.h := max( self.h0, self.h1 );
  }
}
```

Figure 12: Class definitions in Hecate for rendering tree example, optimized with vector data structure. Only key refactorings are listed.

⟨*parallel*⟩ construct in the symbolic traversal. Then, Hecate can verify the absence of inter-dependency between children and generate a parallel scheduling over the list of children. Here, we evaluate the performance of the following variants: 1). Hecate$^{\mathbb{L}}$: sequential linked-list-based traversal 2). Hecate$^{\mathbb{V}}$: sequential vector-based traversal 3). Hecate$^{\mathbb{P}}$: parallel vector-based traversal. Figure 11 summarizes the comparison with Grafter. Here, each line corresponds to one of the variants. The x-axis shows the tree size and the workload is directly adopted from the Grafter paper. The y-axis shows normalized running time over the unfused baseline, averaged over 10 trials.

Linked-List-Based Traversal. Due to limitation of Grafter's static analysis, it only supports linked list for modeling variable-length arrays of children. The Hecate$^{\mathbb{L}}$ variant uses the same linked list data structure, and is able to synthesize a schedule that is semantically equivalent to Grafter's fused traversal. Specifically, Hecate$^{\mathbb{L}}$ achieves competitive performance against Grafter, where both candidates get more than 50% running time reduction over the unfused traversal.

Sequential Vector-Based Traversal. The traversals in real-world compilers like Clang [37] leverage vectors for iterating children. Because the vector-based layout typically leads to better cache locality and reduces the number of dynamic dispatching due to virtual functions, it is crucial for a traversal synthesizer to explore different design choices. However, Grafter does not support vector-based representation due to limitation in its static analysis. On the other

```
traversal layout {
  case Inner{
    iterate cs {
      recur cs;
      ι0;
      ι1;
    }
    ι2;
  }
  case Leaf{...}
}
```

(a) symbolic

```
traversal layout {
  case Inner{
    iterate cs {
      recur cs;
      eval self.h1;
      eval self.w;
    }
    eval self.h;
  }
  case Leaf{...}
}
```

(b) concrete

Figure 13: Symbolic and concrete tree traversals for rendering tree example, optimized with vector data structure. Only key refactorings are listed.

hand, as shown in Figure 12 and Figure 13, it only takes Hecate$^{\mathbb{V}}$ a few lines of changes to refactor a linked-list-based traversal to its vector-based version. In particular, Hecate$^{\mathbb{V}}$ achieves around 70% running time reduction and almost 40% speed-up over Grafter's fused traversal.

Parallel Vector-Based Traversal. As Grafter tacitly assumes that fusion opportunities should be exploited whenever possible, it's designed to reduce the number of tree node visits. This heuristic, despite being effective in some scenarios, may prevent further optimizations and lead to sub-optimal traversals in terms of overall running time.

Consider the fused example shown in Figure 14(b): the fused loop iterates over the children to call a traversal function `c->fusedCalc()` before updating the running maximum for certain values. Assuming that each `c->fusedCalc()` is independent from each other, we can "de-fuse" the for loop into two: as shown in Figure 14(c) the first loop is decomposed into *parallel traversals*, and the second loop updates the running maximum in a sequential fashion. Although the "de-fused" traversal yields a higher number of node visits, it can benefit from parallel execution if the cost of children traversal calls far outweighs the cost of the sequential second visit. This example shows how unconditionally fusing computations might prevent fine-grained optimizations.

As shown in Figure 11, as the tree size grows, the speed-up brought by the parallel variant Hecate$^{\mathbb{P}}$ gradually overcomes its overhead, bringing an additional 23% improvement over the sequential vector-based variant Hecate$^{\mathbb{V}}$.

The evaluation shows that, with minimal effort, Hecate can effectively explore traversals of different design choices.

6.3 Synthesizing Layout Engine in FTL

To show the advantages of our domain-specific encoding, we compare Hecate against FTL [32], a synthesizer specialized for layout engines. In particular, FTL introduces a Prolog-style declarative language for expressing partial schedules with holes. After that, FTL devises a sophisticated synthesis algorithm that leverages Prolog's unification algorithm for effectively generating the schedule as a composition of parallel tree traversals.

Benchmarks. Since FTL is not actively maintained anymore, we can only run it on three variants of attribute grammars (i.e., CSS

```
class Inner: public Box{
public:
  vector<Box*> cs;
}
void Inner::calcWidth() {
  w = w0;
  for (auto c : cs) {
    c->calcWidth();
    w = max( w, c->w );
  }
}
void Inner::calcHeight() {
  h1 = 0;
  for (auto c : cs) {
    c->calcHeight();
    h1 += c->h;
  }
  h = max( h0, h1 );
}
```

(a) unfused version

```
/* class def same as unfused
*
*
*/
void Inner::fusedCalc() {
  w = w0;
  h1 = 0;
  for (auto c : cs) {
    c->fusedCalc();
    w = max( w, c->w );
    h1 += c->h;
  }
  h = max( h0, h1 );
}
```

(b) fused version

```
/* class def same as unfused */
void Inner::fusedCalc() {
  w = w0;
  h1 = 0;
  // parallel
  for (auto c : cs) {
    c->fusedCalc();
  }
  // sequential
  for (auto c : cs) {
    w = max( w, c->w );
    h1 += c->h;
  }
  h = max( h0, h1 );
}
```

(c) "de-fused" version

Figure 14: Pseudo-code class definitions (unfused, fused and "de-fused" versions) for rendering tree example, optimized with vector data structure. Only key refactorings are listed.

Name	# of Rules
CSS-float	192
CSS-margin	178
CSS-full	244

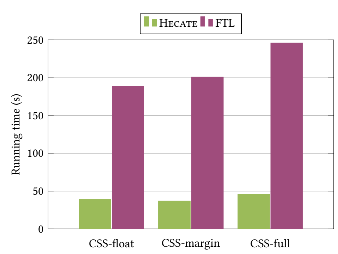


Figure 15: Comparison against FTL: benchmark statistics (left) and results (right).

rules) that are not supported by Grafter: 1) `CSS-float` represents the basic CSS rules together with float rules [6, 8, 10], 2) `CSS-margin` denotes the basic CSS rules together with rules for margin collapse [5, 7, 9], and 3) `CSS-full` is the superset of the previous two and it incorporates the most challenging CSS features such as absolute position, margin collapse, float, and others. Figure 15(left) summaries the statistics of the attribute grammars in terms of number of rules.

Performance. Figure 15(right) shows the results of the comparison and it takes Hecate only a fraction of time in FTL. Specifically, for the `CSS-float` grammar, it takes FTL 189 seconds to synthesize the traversal while it only takes Hecate 39 seconds to finish. As the number of rules grows in `CSS-full`, both tools take a bit longer time, but Hecate is still 5X faster than FTL. To confirm the effectiveness of our domain-specific encoding, we run the general-purpose encoding Hecate$^{\mathbb{G}}$ on all three benchmarks. Hecate$^{\mathbb{G}}$ can not terminate within 30 mins.

This evaluation shows that Hecate can be extended to compute complex CSS semantics supported by real-world layout engines and the domain specific encoding plays a crucial on scaling the tool on those complex benchmarks.

7 RELATED WORK

The closest analog to Hecate in the existing literature is FTL [32]. Like Hecate, FTL synthesizes schedules for browser layout engines; unlike Hecate, FTL translates the layout semantics to a Prolog program, and uses the Prolog kernel to search for schedules. Also unlike Hecate, FTL is specialized to a particular solver, constraint encoding, attribute grammar language, and schedule language; Hecate is considerably more flexible, and its trace language allows it to scale to larger and more complex attribute grammars.

Grafter is another synthesizer for tree traversals. Unlike Hecate, Grafter is based on static analysis, where it generates automata that captures the dependencies indicated between statements and invokes a deterministic algorithm to rewrite and fuse traversals into more compact ones, thus synthesizing new traversals. While Grafter is fast, extending it to new specifications may require extra expert knowledge to devise new tree fusion theories.

Several authors have produced formalizations of browser layout like those used by Hecate to define the layout semantics. Besides those introduced by FTL, Cassius [35] formalizes a subset of browser layout in linear real arithmetic in order to synthesize CSS from examples using an SMT solver, and VizAssert [34] extends that formalization with finitization reductions to support a large subset of the CSS standard, including floating layout, which is widely used in modern web pages but is tricky even for experts to reason about. The Cornipickle [22] project, meanwhile, used first-order modal logic to define visual proproperties of specific web pages. VizAssert later adapted Cornipickle's logic to SMT reasoning. Besides web page layout in particular, there is a rich history of work on constraint-based systems for specifying and synthesizing layouts [1, 4, 23, 44, 48, 52] and on domain-specific languages for describing structured graphics [51] and visual manipulations [17].

Tools for layout problems in web pages form a rich and dynamic topic in the software engineering literature [3, 28–30, 49, 50]. Tools to detect parts of a web page that render differently in different browsers [16, 31, 39] are a large and important subclass of these tools. While these tools are aimed for web page developers (unlike Hecate, which may be used by browser developers), their number demonstrates the challenges that layout bugs impose on practitioners and the importance of the problems Hecate addresses. In fact, practitioners commonly test their web pages against specific instances of browsers and operating systems by loading pages in virtual machine instances [11–13]. The manual inspection that this easy-to-use and widely adopted testing approach requires could be reduced if better tooling reduces the frequency or severity of layout bugs.

Many attribute grammar formalisms [26] assume dynamic scheduling, in contrast to the fully static scheduling presented here. For a large class of attribute grammars, the problem of scheduling an attribute grammar onto a sequence of traversals is known to be NP-hard [19], though polynomial-time scheduling algorithms for restricted classes of grammars exist [32]. However, these restricted classes have not been classified or well-studied.

Constraint solving based on satisfiability modulo theories [33] has become a powerful tool for program analysis as practical, high-performance solvers have become available [2, 18, 21]. Solver-based verification and synthesis tools have a long and rich history in programming languages community [27, 42, 43]. Traditional solver-aided tools use a custom constraint solver or manually translate problems into constraints for a specific existing solver. Solver-aided domain-specific languages [45, 47] instead automatically generate solver constraints based on symbolic execution and custom language extensions. For example, Rosette [46] uses Racket's meta-programming features to provide a high-level interface to several solvers. Hecate is build atop Rosette, but uses its trace language to abstract over the low-level features presented in generic Rosette constraints and significantly improves runtime.

8 CONCLUSION

We propose Hecate, a novel framework for synthesizing tree traversals. The core of Hecate is a *domain-specific* symbolic compilation strategy for tree traversal synthesis that maintains the engineering advantages of solver-aided language, yet achieves better performance. The evaluation shows that Hecate's tree language is expressive as it supports traversals from all Grafter benchmarks and complex features in layout engines. Hecate's domain-specific symbolic compilation is efficient as it achieves 3× speed-up compared to general-purpose symbolic compilation. Finally, Our case analysis shows that Hecate can explore traversals of different design choices with simple modifications.

ACKNOWLEDGMENTS

This work has been supported in part by the ACI OAC–1535191, FMitF CCF-1918027, OIA-1936731, SaTC-1908494, 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, grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032, the Google faculty award, as well as gifts from Adobe, Facebook, Google, Intel, and Qualcomm.

A CASE STUDY: AST

Compilers routinely traverse abstract syntax trees (ASTs) to perform program transformation and validation. The AST benchmark models a simple imperative language with variable assignments, arithmetic expressions, decrement and increment statements, conditional statements, and functions. The benchmark further implements a total of six de-sugaring and optimization passes: 1) de-sugaring decrement statements, 2) de-sugaring increment statements, 3) constant propagation, 4) replacement of variable references to constants, 5) constant folding, and 6) elimination of unreachable branches.

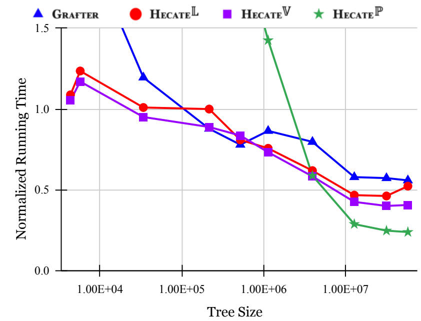


Figure 16: Running time of fused traversals compared to the unfused baseline.

Similar to the RenderTree benchmark, we evaluate the performance of three variants of Hecate: 1). Hecate$^{\mathbb{L}}$: sequential linked-list-based traversal 2). Hecate$^{\mathbb{V}}$: sequential vector-based traversal 3). Hecate$^{\mathbb{P}}$: parallel vector-based traversal. Figure 16 summarizes the comparison with Grafter. Each line corresponds to one of the variants. The x-axis shows the tree size, while the y-axis shows normalized running time over the unfused baseline, averaged over 10 trials.

Overall, the linked-list based traversal Hecate$^{\mathbb{L}}$ achieves around 50% running time reduction compared to unfused baseline, which is similar to Grafter fused traversal. However, the choice of linked lists for representing lists of statements is not so much a necessity as a limitation from Grafter's static analysis. Hecate$^{\mathbb{V}}$ in contrast, lets us replace the underlying data structure with vectors with minimal code modification, leading to a further 10% reduction in running time. Furthermore, Hecate$^{\mathbb{P}}$ is able to take advantage of the data-independency between optimization passes on different AST functions. Although there is ineivitable overhead when the parallel schedules synthesized by Hecate$^{\mathbb{P}}$ are evaluated on smaller trees, the performance gains gradually overcome the overhead, and result in over 75% running time reduction over the unfused baseline.

REFERENCES

[1] 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

[2] 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

[3] 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

[4] 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

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

[6] Bert Bos. 2016. CSS 2.2: Floats. https://tinyurl.com/ssn8rco.

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

[8] Bert Bos, Tantek Çelik, Ian Hickson, and Håkon Wium Lie. 2011. CSS 2.1: Floats. https://tinyurl.com/s67ebaa.

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

[10] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. 1998. CSS 2: Floats. https://tinyurl.com/vbt29em.

[11] Browserling. 2018. https://www.browserling.com/

[12] Browsershots. 2018. http://browsershots.org/

[13] Browserstack. 2018. https://www.browserstack.com/screenshots

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

[15] Kartik Chandra and Rastislav Bodík. 2018. Bonsai: synthesis-based reasoning for type systems. *Proc. ACM Program. Lang.* 2, POPL (2018), 62:1–62:34.

[16] 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

[17] 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

[18] 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

[19] Joost Engelfriet and Gilberto Filè. 1982. Simple multi-visit attribute grammars. *Journal of computer and system sciences* 24, 3 (1982), 283–314. http://doc.utwente.nl/69001/

[20] 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.

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

[22] Sylvain Halle, Nicolas Bergeron, Francis Guerin, and Gabriel Le Breton. 2015. Testing Web Applications Through Layout Constraints. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–8. https://doi.org/10.1109/ICST.2015.7102635

[23] 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

[24] IBM. 2016. The CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer/.

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

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

[27] 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

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

[29] 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

[30] S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond. 2018. 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

[31] 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

[32] 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

[33] 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

[34] Pavel Panchekha, Adam T. Geller, Michael D Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying That Web Pages Have Accessible Layout *(PLDI'18)*. https://doi.org/10.1145/3192366.3192407

[35] 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

[36] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 396–407. https://doi.org/10.1145/2594291.2594339

[37] The Clang Project. 2021. Clang: a C language family frontend for LLVM. https://clang.llvm.org.

[38] The Servo Parallel Browser Engine Project. 2018. Rendering issue of styled buttons. https://github.com/servo/servo/issues/18991.

[39] 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

[40] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: a framework for analyzing and fusing general recursive tree traversals. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 76:1–76:30.

[41] 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.

[42] 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

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

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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

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

[52] 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