CSS Minification via Refactoring and Constraint Solving

MATTHEW HAGUE, Royal Holloway, University of London
ANTHONY W. LIN, University of Oxford
CHIH-DUO HONG, University of Oxford

Minification is a widely accepted technique which aims at reducing the size of the code transmitted over the web. This paper concerns the problem of semantic-preserving minification of Cascading Style Sheets (CSS) — the de facto language for styling web documents — based on refactoring similar rules.

The cascading nature of CSS makes the semantics of CSS files sensitive to the ordering of rules in the file. To automatically identify rule refactoring opportunities that best minimise file size, we reduce the refactoring problem to a problem concerning "CSS-graphs", i.e., node-weighted bipartite graphs with a dependency ordering on the edges, where weights capture the number of characters.

Constraint solving plays a key role in our approach. Transforming a CSS file to a CSS-graph problem requires us to extract the dependency ordering on the edges (an NP-hard problem), which requires us to solve the selector intersection problem. To this end, we provide the first full formalisation of CSS3 selectors (the most stable version of CSS) and reduce their selector intersection problem to satisfiability of quantifier-free integer linear arithmetic, for which highly-optimised SMT-solvers are available. To solve the above NP-hard graph optimisation problem, we show how Max-SAT solvers can be effectively employed. We have implemented our rule refactoring algorithm, and tested it against ~ 70 real-world examples (including top 20 most popular websites). We also used our benchmarks to compare our tool against six well-known minifiers (which implement other optimisations). Our experiments suggest that our tool produced larger savings. A substantially better minification rate was shown when our tool is used together these minifiers.

CCS Concepts: •**Software and its engineering → Syntax; Semantics; Software maintenance tools;** •**Theory of computation → Automated reasoning; Tree languages;**

Additional Key Words and Phrases: Cascading Style Sheets, Web-optimisation, Semantics, Max-SAT

ACM Reference format:
Matthew Hague, Anthony W. Lin, and Chih-Duo Hong. 2017. CSS Minification via Refactoring and Constraint Solving. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2017), 66 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Minification (Souders 2007, Chapter 12) is a widely accepted technique in the web programming literature that aims at decreasing the size of the code transmitted over the web, which can directly improve the response time performance of a website. Page load time is of course crucial for users' experience, which impacts the performance of an online business and is increasingly being included as a ranking factor by search engines (Slegg 2017). Minification bears a resemblance to traditional code compilation. In particular, it is applied only *once* right before deploying the website (therefore, its computation time does *not* impact the page load time). However, they differ in at least two ways. Firstly, the source and target languages for minification are the same (high-level) languages. The code to which minification can be applied is typically JavaScript or CSS, but it can also be HTML, XML, SVG, etc. Secondly, minification applies various semantic-preserving transformations with the objective of reducing the size of the code.

2017. 0164-0925/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

This paper concerns the problem of minifying CSS (Cascading Style Sheets), which is the de facto language for styling web documents (HTML, XML, etc.) as developed and maintained by World Wide Web Constortium (W3C) (Atkins Jr. et al. 2017). A CSS file consists of a list of CSS *rules*, each containing a list of *selectors* — each selecting nodes in the *Document Object Model* (DOM), which is a tree structure representing the document — and a list of *declarations*, each assigning values to selected nodes' display attributes (e.g. `blue` to the property `color`). Real-world CSS files can easily have many rules (in the order of magnitude of 1000), e.g., see the statistics for popular sites (Alexa Internet 2017) on http://cssstats.com/. As Souders wrote in his book (Souders 2007), which is widely considered as the bible of web performance optimisation:

> The greatest potential for [CSS file] size savings comes from optimizing CSS — merging identical classes, removing unused classes, etc. This is a complex problem, given the order-dependent nature of CSS (the essence of why it's called *cascading*). This area warrants further research and tool development.

Hitherto, more and more CSS minifiers have been developed. These include `YUI Compressor` (Sha and Contributors 2014), `cssnano` (Briggs and Contributors 2015), `minify` (Clay and Contributors 2017), `clean-css` (Pawlowicz 2017), `csso` (Dvornov and Contributors 2017), and `cssmin` (Bleuzen 2017), to name a few. Such CSS minifiers apply various syntactic transformations, typically including removing whitespace characters and comments, and using abbreviations (e.g. `#f60` instead of `#ff6600` for the colour orange).

In this paper, we propose a new class of CSS file size-reducing transformations based on *refactoring* (Fowler 1999) similar or duplicate rules in the CSS file by merging or combining them (thereby removing repetitions across multiple rules in the file) without affecting how a document is rendered. To illustrate the type of refactoring that we focus on in this paper (a more detailed introduction is given in Section 2), consider the following simple CSS file.

```
#a { color:red; font-size:large }
#c { color:green }
#b { color:red; font-size:large }
```

The selector `#a` selects a node with *ID* `a` (if exists). Note that the first and the third rules above have the same property declarations. Since one ID can be associated with at most one node in a DOM-tree, we can merge these rules resulting in the following equivalent file.

```
#a, #b { color:red; font-size:large }
#c { color:green }
```

Identifying such a refactoring opportunity in general is, however, non-trivial since a CSS file is sensitive to the ordering of rules that may match the same node, i.e., the problem mentioned in Souders' quote above. For example, let us assume that the three selectors `#a`, `#b`, and `#c` in our CSS example instead were `.a`, `.b`, and `.c`, respectively. The selector `.a` selects nodes with *class* `a`. Since a class can be associated with multiple nodes in a DOM-tree, the above refactoring could change how a page is displayed and therefore would not be valid, e.g., consider a page with an element that has the classes `.b` and `.c`, which would be displayed as red by the original file, but as green by the file after applying refactoring. Despite this, in this case we would still be able to refactor the *subrules* of the first and the third rules (associated with the `font-size` property) resulting in the following smaller equivalent file:

```
.a { color:red }
.c { color:green }
.b { color:red }
.a, .b { font-size:large }
```

This example suggests that to solve the CSS rule refactoring problem in a general way, we need to have a means of checking whether two given selectors may *intersect*, i.e., select the same node in *some* document tree. In addition, such a refactoring-based method (if can be done efficiently) could be used to perform CSS minification by a simple greedy algorithm which at each step finds and applies a rule refactoring that best minimises the CSS file until no rule refactoring can further reduce the file size. The key difficulty of course is how to find such a refactoring and how to do it efficiently, both of which we fully address in this paper. It is also equally important to ask whether rule refactoring can have a non-negligible impact in CSS minification. Our paper provides a positive answer to this question.

1.1 Contributions

We first formulate a *general class of semantic-preserving transformations* on CSS files that captures the above notion of refactoring which merges and combines similar rules in a CSS file. Such a program transformation has a clean graph-theoretic formulation (see Section 4). Loosely speaking, a CSS rule corresponds to a biclique (complete bipartite graph) B, whose edges connect nodes representing selectors and nodes representing property declarations. Therefore, a CSS file F corresponds to a sequence of bicliques that covers all of the edges in the bipartite graph $\mathcal{G}$ which is constructed by taking the (non-disjoint) union of all bicliques in F. Due to the cascading nature of CSS, the file F also gives rise to an (implicit) ordering $\prec$ on the edges of $\mathcal{G}$. Therefore, any new CSS file F' that we produce from F must also be a valid covering of the edges of $\mathcal{G}$ and respect the order $\prec$. As we will see, the above notion of refactoring corresponds to a pair $(\overline{B}, j)$ of a new rule $\overline{B}$ and a position j in the file, and that applying this refactoring entails inserting $\overline{B}$ in position j and removing all redundant nodes (i.e. either a selector or a property declaration) in rules at position $i < j$.

Several questions remain. First is how to compute the edge order $\prec$. The core difficulty of this problem is to determine whether two CSS selectors can be matched by the same node in some document tree (a.k.a. the *selector intersection problem*). Second, among the multiple potential refactoring opportunities, how do we automatically compute a rule refactoring that best minimises the size of the CSS file. We provide solutions to these questions in this paper.

Computing the edge order $\prec$. In order to handle the selector intersection problem, we first provide a complete formalisation of CSS3 selectors (Çelik et al. 2011) (currently the most stable version of CSS selectors). We then give a polynomial-time reduction from the selector intersection problem to satisfiability over quantifier-free theory of integer linear arithmetic, for which highly optimised SMT-solvers (e.g. Z3 (de Moura and Bjørner 2008)) are available. To achieve this reduction, we provide a chain of polynomial-time reductions. First, we develop a new class of *automata over data trees* (Bojańczyk 2010), called *CSS automata*, which can capture the expressivity of CSS selectors. This reduces the selector intersection problem to the language intersection problem for CSS automata. Second, unlike the case for CSS selectors, the languages recognised by CSS automata enjoy closure under intersection. [`.b .a` and `.c .a` are individually CSS selectors, however their conjunction is not a CSS selector[1].] This reduces language intersection of CSS automata to language non-emptiness of CSS automata. To finish this chain of reductions, we provide a reduction from the problem of language non-emptiness of CSS automata to satisfiability over quantifier-free theory of integer linear arithmetic. The last reduction is technically involved and requires insights from logic and automata theory, which include several small model lemmas (e.g. the sufficiency of considering trees with a small number of node labels that can be succinctly encoded). This is despite the fact that CSS selectors may force the smallest satisfying tree to be exponentially big.

[1]The conjunction can be expressed by the *selector group* `.b .c .a, .c .b .a, .b.c .a`.

Formulation of the "best" refactoring and how to find it. Since our objective is to minimise file size, we may equip the graph $\mathcal{G}$ by a *weight function* `wt` which maps each node to the number of characters used to define it (recall that a node is either a selector or a property declaration). Hence, the function `wt` allows us to define the size of a CSS file F (i.e. a covering of $\mathcal{G}$ respecting $\prec$) by taking the sum of weights $\texttt{wt}(v)$ ranging over all nodes v in F. The goal, therefore, is to find a refactoring $(\overline{B}, j)$ of F that produces F' with a minimum file size. We show how this problem can be formulated as a (partially weighted) Max-SAT instance (Argelich et al. 2016) in such a way that several existing Max-SAT solvers (including Z3 (Bjørner and Narodytska 2015) and MaxRes (Narodytska and Bacchus 2014)) can handle it efficiently. This Max-SAT encoding is non-trivial: the naive encoding causes Max-SAT solvers to require prohibitively long run times even on small examples. A naive encoding would allow the Max-SAT solver to consider any rule constructed from the nodes of the CSS file, and then contain clauses that prohibit edges that do not exist in the original CSS file (as these would introduce new styling properties). Our encoding forces the Max-SAT solver to only consider rules that do not introduce new edges. We do this by enumerating all *maximal* bicliques in the graph $\mathcal{G}$ (maximal with respect to set inclusion) and developing an encoding that allows the Max-SAT solver to only consider rules that are contained within a maximal biclique. We employ the algorithm from (Kayaaslan 2010) for enumerating all maximal bicliques in a bipartite graph, which runs in time polynomial in the size of the input and output. Therefore, to make this algorithm run efficiently, the number of maximal bicliques in the graph $\mathcal{G}$ cannot be very large. Our benchmarking (using $\sim$ 70 real-world examples including those from the top 20 websites (Alexa Internet 2017)) suggests that this is the case for graphs $\mathcal{G}$ generated by CSS files (with the maximum ratio between the number of bicliques and the number of rules being 2.05). Our experiments suggest that the combination of the enumeration algorithm of (Kayaaslan 2010) and Z3 (Bjørner and Narodytska 2015) makes the problem of finding the best refactoring for CSS files practically feasible.

Experiments. We have implemented our CSS minification algorithm in the tool SATCSS which greedily searches for and applies the best refactoring opportunity to a given CSS file until no more refactoring can reduce file size. This will be made available in the supplementary material. The source code and a working disk image are also available from the following URLs:

https://github.com/matthewhague/sat-css-tool
http://www.cs.rhul.ac.uk/home/hague/sat-css-tool.img.txz

Our tool utilises Z3 (Bjørner and Narodytska 2015; de Moura and Bjørner 2008) as a backend solver for Max-SAT and SMT over integer linear arithmetic. We have tested our tool on $\sim$ 70 examples from real-world websites (including those from the top 20 websites (Alexa Internet 2017)) with promising experimental results. We found that SATCSS (which only performs refactoring) yields larger savings on our benchmarks in comparison to six popular CSS minifiers (Slant 2017), which support many other optimisations but not our notion of refactoring. Moreover, our experiments also suggest that an even better minification performance can be achieved if we apply our tool in conjunction with these minifiers. More precisely, when run individually, SATCSS achieved savings with a third quartile of 6.77% and a median value of 3.91%. The six mainstream minifiers achieved savings with third quantiles and medians up to 5.87% and 3.24%, respectively. When we run our tool after running any one of these minifiers, the third quartile of the savings can be increased to 8.30% and the median to 5.11%. The additional gains obtained by our tool on top of the six minifiers (as a percentage of the original file size) have a third quartile of 5.19% and a median value of 3.22%. Moreover, the ratios of the percentage of savings made by SATCSS to the percentage of savings made by the six minifiers have third quartiles of at least 144% and medians of at least 63%.

These figures clearly indicate a substantial proportion of extra space savings made by SATCSS. See Table 1 and Figure 9 for more detailed statistics.

1.2 Organisation

Section 2 provides a gentle and more detailed introduction (by example) to the problem of CSS refactoring. Preliminaries (notation, and basic terminologies) can be found in Section 3. In Section 4 we formalise the refactoring problem in terms of CSS-graphs. In Section 5, we provide a formalisation of CSS3 selectors and an outline of an algorithm for solving their intersection problem, from which we can extract the edge order of a CSS-graph this is required when reducing the rule refactoring problem to the edge-covering problem of CSS-graphs. Since the algorithm solving the selector intersection problem is rather intricate, we dedicate one full section (Section 6) for it. In Section 7 we show how Max-SAT solvers can be exploited to solve the refactoring problem of CSS-graphs. We describe our experimental results in Section 8. In Section 9 we give a detailed discussion of related work. Finally, we conclude in Section 10. Other missing details can be found in the appendix. The Python code and benchmarks for our tool have been included in the supplementary material, with a brief user guide. A full artefact, including virtual machine image will be included when appropriate. These are presently available from the URLs above.

2 CSS REFACTORING (BY EXAMPLE)

In this section, we provide a gentler and more detailed introduction to CSS refactoring by merging and combining rules, while elaborating on the difficulties of the problem. We assume basic familiarity with HTML and CSS. Each node in the DOM-tree is labeled by precisely one tag name (e.g. `div`), one or more *class* attributes, and at most one *ID* attribute (whose value is unique). A CSS file consists of a sequence of *rules*, each of the form

$$\textit{selectors} \; \{ \; \textit{declarations} \; \}$$

where *selectors* contains one or more (node)-selectors (separated by commas) and *declarations* contains a sequence of (visual) property declarations (separated by semicolons). Figure 1 contains a simple CSS file (with four rules). The semantics of a rule is simple: if a node can be matched by at least one of the selectors, then label the node by all the visual properties in the rule. E.g., the second rule in Figure 1 reads: if a node has class `fruit` (the `.` notation) or the ID `broccoli` (the `#` notation), then assign the labels `color:red` and `font-size:large` to the node. The sequence of rules in a file is applied to a node v in a "cascading" fashion (hence the name Cascading Style Sheets). That is, read the rules from top to bottom and check if v can be matched by at least one selector in the rule. If so, assign the properties to v (perhaps overriding previously assigned properties, e.g., color) provided that the selectors matching v in the current rule have higher "values" (a.k.a. *specificities*) than the selectors previously matching v. Intuitively, the specificity of a selector (Çelik et al. 2011) can be calculated by taking a weighted sum of the number of classes, IDs, tag names, etc. in the selector, where IDs have higher weights than classes, which in turn have higher weights than tag names. For example, let us apply this CSS file to a node matching `.fruit`

```
#apple { color:blue; font-size:small }
.fruit, #broccoli { color:red; font-size:large }
#orange { color:blue }
#tomato { color:red; font-size:large;
          background-color:lightblue }
```

Fig. 1. A simple example of a CSS file.

and `#apple`. In this case, the selectors in the first (`#apple`) and the second rules (`.fruit`) are applicable. However, since `#apple` has a higher specificity than `.fruit`, the node gets labels `color:blue` and `font-size:small`.

Two syntactically different CSS files could be "semantically equivalent" in the sense that, on each DOM tree T, both CSS files will precisely yield the same tree T' (which annotates T with visual information). In this paper, we only look at semantically equivalent CSS files that are obtained by refactoring via merging/combining similar rules. Given a CSS rule R, we say that it is *subsumed* by a CSS file F if each possible selector/property combination in R occurs in some rule in F. Notice that a rule can be subsumed in a CSS file F without occurring in F as one of the rules. For example, the rule R_1

```
    .fruit, #broccoli, #tomato { color:red; font-size:large }
```

is subsumed in our CSS file example (not so if `background-color:lightblue` were added to R_1). A *refactoring opportunity* consists of a CSS rule subsumed in the CSS file and a position in the file to insert the rule into. An example of a refactoring opportunity in our CSS file example is the rule R_1 and the end of the file as the insertion position. This results in a new (bigger) CSS file. We then "trim" this resulting CSS file by eliminating "redundant" subrules. For example, the second rule is redundant since it is completely contained in the new rule R_1. Also, notice that the subrule:

```
    #tomato { color:red; font-size:large }
```

of the fourth rule in the original file is also completely redundant since it is contained in R_1. Removing these, we obtain the following trimmed version of the CSS file:

```
    #apple { color:blue; font-size:small }
    #orange { color:blue }
    #tomato { background-color:lightblue }
    .fruit, #broccoli, #tomato { color:red; font-size:large }
```

This new CSS file contains fewer bytes. We may apply another round of refactoring by inserting the rule

```
    #apple, #orange { color:blue }
```

at the end of the second rule (and trim). The resulting file is even smaller. Computing such refactoring opportunities (i.e. which yields maximum space saving) is in fact a difficult problem.

Two remarks are in order. Not all refactoring opportunities yield a smaller CSS file. For example, if `#tomato` is replaced by the fruit vegetable

```
#vigna_unguiculata_subsp_sesquipedalis
```

the first refactoring above would have resulted a larger file, which we should *NOT* apply. The second remark is related to the issue of *order dependency*. Suppose we add the selector `.vegetable` to the third rule in the original file, resulting in the following rule

```
    .vegetable, #orange { color:blue }
```

Any node labeled by `.fruit` and `.vegetable` but no IDs will be assigned `color:blue`. However, performing the first refactoring above yields a CSS file which assigns `color:red` to fruit vegetables, i.e., *not* equivalent to the original file. To take the issue of order dependency, we need to account for selectors specificity and whether two selectors with the same specificity can *intersect* (i.e. can be matched by a node in some DOM tree). Although for our examples the problem of selector intersection is quite obvious, this is not the case for CSS selectors in general, e.g., the following two selectors from a real-world CSS example found on The Guardian website

```
    .commercial--masterclasses .lineitem:nth-child(4)
    .commercial--soulmates:nth-child(n+3)
```

intersect, while changing `n+3` to `2n+3` would yield the intersection empty!

3 PRELIMINARIES

3.1 Maths

As usual, $\mathbb{Z}$ denotes the set of all integers. We use $\mathbb{N}$ to denote the set $\{0, 1, \ldots, \}$ of all natural numbers. Let $\mathbb{N}_{>0} = \mathbb{N} \setminus \{0\}$ denote the set of all positive integers. For an integer x we define $|x|$ to be the absolute value of x. For two integers i, j, we write $[i, j]$ to denote the set $\{i, \ldots, j\}$. Similar notations for open intervals will also be used for integers (e.g. (i, j) to mean $\{i + 1, \ldots, j - 1\}$). In the sequel, for a set S, we will use S^* (resp. S^+) to denote the set of sequences (resp. non-empty sequences) of elements from S. When the meaning is clear, if S is a singleton $\{s\}$, we will denote $\{s\}^*$ (resp. $\{s\}^+$) by s^* (resp. s^+). Given a (finite) sequence $\sigma = s_1, \ldots, s_n$, $i \in [0, n]$, and a new element s, we write $\sigma[s \rightarrow i]$ to denote the new sequence $s_1, \ldots, s_i, s, s_{i+1}, \ldots, s_n$, i.e., inserting the element s right after the position i in σ.

3.2 Trees

When dealing with document trees, it is standard to deal with *rooted unranked ordered trees*. More formally, a *tree domain* is a set $D \subseteq (\mathbb{N}_{>0})^*$ of *nodes* that is both *prefix-closed* and *preceding-sibling closed*, i.e., $\eta\iota \in D$ implies both $\eta \in D$, and $\eta\iota' \in D$ for all $\iota' < \iota$. Observe, we write η for a tree node (element of D) and ι for an element of $\mathbb{N}_{>0}$. A Σ*-labelled tree* is a pair $T = (D, \lambda)$ where D is a tree domain, and $\lambda : D \rightarrow \Sigma$ is a labelling function of the nodes of T. Next we recall terminologies for relationships between nodes in trees. To avoid notational clutter, we deliberately choose notation that resembles the syntax of CSS. In the following, take $\eta, \eta' \in D$. We write $\eta \gg \eta'$ if η is a (strict) ancestor of η', i.e., there is some $\eta'' \in \mathbb{N}_{>0}^+$ such that $\eta' = \eta\eta''$. We write $\eta > \eta'$ if η is the parent of η', i.e., there is some $\iota \in \mathbb{N}_{>0}$ such that $\eta' = \eta\iota$. We write $\eta + \eta'$ if η is the direct preceding sibling of η', i.e., there is some η'' and $\iota \in \mathbb{N}_{>0}$ such that $\eta = \eta''(\iota - 1)$ and $\eta' = \eta''\iota$. We write $\eta \sim \eta'$ if η is a preceding sibling of η', i.e., there is some η'' and $\iota, \iota' \in \mathbb{N}_{>0}$ with $\iota < \iota'$ such that $\eta = \eta''\iota$ and $\eta' = \eta''\iota'$.

3.3 Max-SAT

In this paper, we will present a reduction to partial weighted Max-SAT (Argelich et al. 2016). Partial weighted Max-SAT formulas are boolean formulas in CNF with *hard constraints* (a.k.a. clauses that must be satisfied) and *soft constraints* (a.k.a. clauses that may be violated with a specified cost or weight). A minimal-cost solution is the goal. Note that our clauses will not be given in CNF, but standard satisfiability-preserving conversions to CNF exist (e.g. see (Bradley and Manna 2007)) which straightforwardly extends to partial weighted Max-SAT.

We will present a Max-SAT problems in the following form (Π_H, Π_S) where

- Π_H are the hard constraints – that is, a set of boolean formulas that *must* be satisfied – and
- Π_S are the soft constraints – that is a set of pairs (φ, ω) where φ is a boolean formula and $\omega \in \mathbb{N}$ is the weight of the constraint.

Intuitively, the weight of a soft constraint is the cost of not satisfying the constraint. The partial weighted Max-SAT problem is to find an assignment to the boolean variables that satisfies all hard constraints and minimises the sum of the weights unsatisfied soft constraints.

3.4 Existential Presburger Arithmetic

During the paper, we present several encodings into *existential Presburger arithmetic*, also known as the *quantifier-free theory of integer linear arithmetic*. Here, we use extended existential Presburger formulas $\exists x_1, \ldots, x_k.\varphi$ where φ is a boolean combination of expressions $\sum_{i=1}^{k} a_i x_i \sim b$ for constants $a_1, \ldots, a_k, b \in \mathbb{Z}$ and $\sim \in \{\leq, \geq, <, >, =\}$ with constants represented in binary. A formula is satisfiable if there is an assignment of a non-negative integer to each variable $x_1, \ldots, x_k$ such that φ is satisfied. For example, a simple existential Presburger formula is shown below

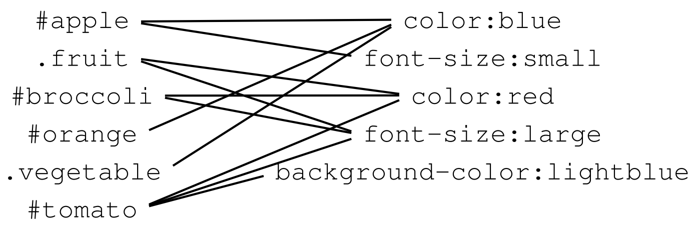
$$\exists x, y, z.x < 2y + z$$

which is satisfied by any assignment to the variables x, y, and z such that $x < 2y+z$. The assignment $x = 2, y = 3, z = 0$ is one such satisfying assignment.

It is well-known that satisfiability of existential Presburger formulas is NP-complete even with the above extensions (cf. (Scarpellini 1984)). Such problems can be solved efficiently by SMT solvers such as Z3.

4 FORMAL DEFINITION OF CSS REFACTORING/MINIFICATION

The semantics of a CSS file can be formally modelled as a CSS-graph. A *CSS-graph* is a 5-tuple $\mathcal{G} = (S, P, E, \prec, \texttt{wt})$, where (S, P, E) is a bipartite graph (i.e. the set V of vertices is partitioned into two sets S and P with $E \subseteq S \times P$), $\prec \subseteq E \times E$ gives the order dependency on the edges, and $\texttt{wt} : S \cup P \to \mathbb{Z}_{>0}$ is a weight function on the set of vertices. Vertices in S are called *selectors*, whereas vertices in P are called *properties*. For example, the CSS graph corresponding to the CSS file in Figure 1 with the selector `.vegetable` added to the third rule is the following bipartite graph



such that the weight of a node is $1+$ (length of the text), e.g., $\texttt{wt}(\texttt{\#orange}) = 1 + 7 = 8$. The reason for the extra $+1$ is to account for the selector/property separators (i.e. commas or semi-colons), as well as the character `'{'` (resp. `'}'`) at the end of the sequence of selectors (resp. properties). We refer to $\prec$ as the *edge order* and it intuitively states that one edge should appear strictly before the other in any CSS file represented by the graph. In this case we have $(\texttt{.fruit}, \texttt{color:red}) \prec (\texttt{.vegetable}, \texttt{color:blue})$ because any node labeled by `.fruit` and `.vegetable` but no IDs should be assigned the property `color:blue`. There are no other orderings since each node can have at most one ID[2] and `.fruit` and `.vegetable` are the selectors of the lowest specificity in the file. More details on how to compute $\prec$ from a CSS file are given in Section 5.4.

A *biclique* in $\mathcal{G}$ is a complete bipartite subgraph, i.e., a pair $B = (X, Y)$ of a nonempty set $X \subseteq S$ of selectors and a nonempty set $Y \subseteq P$ of properties such that $X \times Y \subseteq E$ (i.e. each pair of a selector and a property in the rule is an edge). A *(CSS) rule* is a pair $\overline{B} = (B, \lhd)$ of a biclique and a total

[2]Strictly speaking, this is only true if we are only dealing with namespace `html` (which is the case almost always in web programming and so is a reasonable assumption unless the user specifies otherwise). A node could have multiple IDs, each with a different namespace. See Section 5.

order $\lhd$ on the set of properties. The reason for the order on the properties, but not on the selectors, is illustrated by the following example of a CSS rule:

```
.a, .b { color:red; color:rgba(255,0,0,0.5) }
```

That is, nodes matching `.a` *or* `.b` are assigned a semi-transparent red with solid red being defined as a *fallback* when the semi-transparent red is not supported by the document reader. Therefore, swapping the order of the properties changes the semantics of the rule, but swapping the order of the selectors does not. In the sequel, we often denote a rule as $(X, \overline{Y})$ where $\overline{Y} = \{p_i\}_{i=1}^m$ if $Y = \{p_1, \ldots, p_m\}$ and $p_1 \lhd \cdots \lhd p_m$.

A *covering* C of $\mathcal{G}$ is a sequence of rules that *covers* $\mathcal{G}$ (i.e. the union of all the edges in C equals E). Given an edge $e \in E$, the *index* $\text{index}(e)$ of e is defined to be the index of the *last* rule in the sequence C that contains e. We say that C is *valid* if, for all two edges $e = (s, p), e' = (s', p')$ in E with $e \prec e'$, either of the following holds:

- $\text{index}(e) < \text{index}(e')$
- $\text{index}(e) = \text{index}(e')$ and, if $(X, \{p_i\}_{i=1}^m)$ is the rule at position $\text{index}(e)$ in C, it is the case that $p = p_j$ and $p' = p_k$ with $j \leq k$.

This *last-occurrence* semantics reflects the cascading aspect of CSS. To relate this to the world of CSS, the original CSS file F may be represented by a CSS-graph $\mathcal{G}$, but F also turns out to be a valid covering of $\mathcal{G}$. In fact, the set of valid coverings of $\mathcal{G}$ correspond to CSS files that are equivalent (up to reordering of selectors and property declarations) to the original CSS file.

To define the optimisation problem, we need to define the weight of a rule and the weight of a covering. To this end, we extend the function `wt` to rules and coverings by summing the weights of the nodes. More precisely, given a rule $\overline{B} = (X, \overline{Y})$, define

$$\mathtt{wt}(\overline{B}) = \sum_{w \in X \cup \overline{Y}} \mathtt{wt}(w).$$

Similarly, given a covering $C = \{\overline{B}_i\}_{i=1}^m$, the weight $\mathtt{wt}(C)$ of C is $\sum_{i=1}^m \mathtt{wt}(\overline{B}_i)$. It is easy to verify that the weight of a rule (resp. covering) corresponds to the number of non-whitespace characters in a CSS rule (resp. file). The *minification problem* is, given a CSS-graph $\mathcal{G}$, compute a valid covering with the minimum weight.

(Optimal) Refactoring Problem. Given a CSS-graph $\mathcal{G}$ and a covering C, we define the *trim* $C_\downarrow$ of C to be the covering C' obtained by removing from each rule $\overline{B} = (X, \overline{Y})$ (say at position i) in C all nodes $v \in X \cup \overline{Y}$ that are not incident to at least one edge e with $\text{index}(e) = i$ (i.e. the last occurrence of e in C). Such nodes v may be removed since they do not affect the validity of the covering C. The trim $C_\downarrow$ can be computed from C and $\mathcal{G}$ in polynomial time in a straightforward way. [More precisely, first build a hashmap for the index function. Second, go through all rules $\overline{B}$ in the covering C, each node v in $\overline{B}$, and each edge e incident with v checking if at least one such e satisfies $\text{index}(e) = i$, where $\overline{B}$ is the ith rule in C.] Note that $C_\downarrow$ is uniquely defined given C. We define a *refactoring opportunity* to be a pair $(\overline{B}, j)$ of rule $\overline{B}$ and a number $j \in (0, |C|)$ such that $C[\overline{B} \to j]$ is a valid covering of $\mathcal{G}$. The *result* of applying this refactoring opportunity is the covering $C[\overline{B} \to j]_\downarrow$ obtained by trimming $C[\overline{B} \to j]$. The *refactoring problem* can be defined as follows: given a CSS-graph $\mathcal{G}$ and a valid covering C, find a refactoring opportunity that results in a covering with the minimum weight. This refactoring problem is NP-hard even in the non-weighted version (Peeters 2003; Yannakakis 1978).

5 CSS SELECTOR FORMALISATION AND ITS INTERSECTION PROBLEM

In this section, we will show how to efficiently compute a CSS-graph $\mathcal{G} = (S, P, E, \lessdot, \mathtt{wt})$ from a given CSS file with the help of a fast solver of quantifier-free theory of integer linear arithmetic, e.g., Z3 (de Moura and Bjørner 2008). The key challenge is how to extract the order dependency $\lessdot$ from a CSS file, which requires an algorithm for the *(selector-)intersection problem*, i.e., to check whether two given selectors can be matched by the same element in *some* document. To address this, we provide a full formalisation of CSS3 selectors (Çelik et al. 2011) and a fast algorithm for the intersection problem. Since our algorithm for the intersection problem is technically very involved, we provide a short intuitive explanation behind the algorithm in this section and leave the details to Section 6.

5.1 Definition of Document Trees

We define the semantics of CSS3 in terms of Document Object Models (DOMs), which we also refer to as document trees. The reader may find it helpful to recall the definition of trees from Section 3.2.

A document tree consists of a number of elements, which in turn may have sub-elements as children. Each node has a *type* consisting of an element name (a.k.a. tag name) and a namespace. For example, an element `p` in the default `html` namespace is a paragraph. Namespaces commonly feature in programming languages (e.g. C++) to allow the use of multiple libraries whilst minimising the risk of overloading names. For example, the HTML designers introduced a `div` element to represent a section of an HTML document. Independent designers of an XML representation of mathematical formulas may represent division using elements also with the name `div`. Confusion is avoided by the designers specifying a namespace in addition to the required element names. In this case, the HTML designers would introduce the `div` element to the `html` namespace, while the mathematical `div` may belong to a namespace `math`. Hence, there is no confusion between `html:div` and `math:div`. [As an aside, note that an HTML file may contain multiple namespaces, e.g., see (Hickson et al. 2014).]

Moreover, nodes may also be labelled by attributes, which take string values. E.g. an HTML `img` element has a `src` attribute specifying the source of the image. Finally, a node may be labelled by a number of *pseudo-classes*. For example `:enabled` means that the node is enabled and the user may interact with it. The set of pseudo-classes is fixed by the CSS specification.

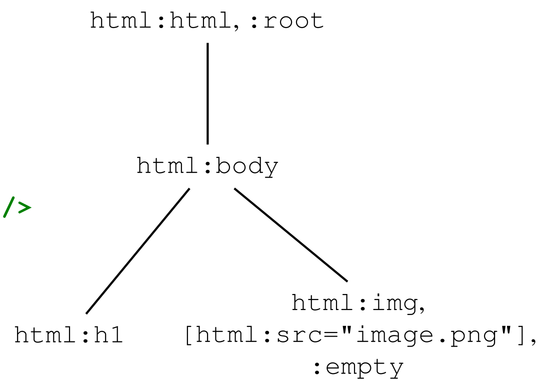
We first give an example before giving the formal definition.

5.1.1 Example. Consider the HTML file in Figure 2a consisting of a heading "Page Title" and a single image. This file can be represented by the tree in Figure 2b. In this tree, each node is first labelled by its type, which consists of its namespace and its element name. In all cases, the namespace is the `html` namespace, while the element names are exactly as in the HTML file. In addition, we label each node with the attributes and pseudo-classes with which they are labelled. The `html:html` node is labelled with `:root` since it is the root element of the tree. The `html:img` node is labelled with the attribute `html:src` with value `image.png` as well as the pseudo-class `:empty` indicating that the node has no contents. The `html:h1` node however is not labelled `:empty` since it contains the text "Page Title" which is not matchable by any selector. Hence, a node with no child may still be non-empty.

5.1.2 Formal Definition. In the formal definition below we permit a possibly infinite set of element, namespace, and attribute names. CSS stylesheets are *not* limited to HTML documents (which have a fixed set of element names, but not attribute names since you can create custom `data-*` attributes), but they can also be applied to documents of other types (e.g. XML) that

```html
<html>
  <body>
    <h1>Page Title</h1>
    <img src="image.png"/>
  </body>
</html>
```

(a) An example HTML file.



(b) A DOM tree representation of the HTML file.

Fig. 2. An HTML file and its representation as a DOM tree.

permit custom element names, namespaces, and attribute names. Thus, the sets of possible names *cannot* be fixed to finite sets from the point of view of CSS selectors, which may be applied to any document.

We denote the set of *pseudo-classes* as

$$P = \left\{ \begin{array}{c} \texttt{:link}, \texttt{:visited}, \texttt{:hover}, \texttt{:active}, \texttt{:focus}, \texttt{:target}, \\ \texttt{:enabled}, \texttt{:disabled}, \texttt{:checked}, \texttt{:root}, \texttt{:empty} \end{array} \right\} .$$

Then, given a possibly infinite set of *namespaces* NS, a possibly infinite set of *element names* E, a possibly infinite set of *attribute names* A, and a finite alphabet[3] Γ containing the special characters ␣ and - (space and dash respectively), a *document tree* is a Σ-labelled tree (D, λ), where

$$\Sigma := \left(\mathrm{NS} \times E \times \mathcal{F}_{\mathrm{fin}}(\mathrm{NS} \times A, \Gamma^*) \times 2^P\right) .$$

Here the notation $\mathcal{F}_{\mathrm{fin}}(\mathrm{NS} \times A, \Gamma^*)$ denotes the set of partial functions from $(\mathrm{NS} \times A)$ to Γ^* whose domain is finite. In other words, each node in a document tree is labeled by a namespace, an element, a function associating a finite number of namespace-attribute pairs with attribute values (strings), and zero or more of the pseudo-classes. For a function $f_A \in \mathcal{F}_{\mathrm{fin}}(\mathrm{NS} \times A, \Gamma^*)$ we say $f_A(s, a) = \perp$ when f_A is undefined over $s \in \mathrm{NS}$ and $a \in A$, where $\perp \notin \Gamma^*$ is a special undefined value. Furthermore, we assume special attribute names $\texttt{class}, \texttt{id} \in A$ that will be used to attach classes (see later) and IDs to nodes.

When $\lambda(\eta) = (s, e, f_A, P)$ we write

$$\begin{array}{rcl} \lambda_{\mathrm{S}}(\eta) & = & s, \\ \lambda_{\mathrm{E}}(\eta) & = & e, \\ \lambda_{\mathrm{A}}(\eta) & = & f_A, \text{ and} \\ \lambda_{\mathrm{P}}(\eta) & = & P. \end{array}$$

In the sequel, we will use the following standard XML notation: for an element name e, a namespace s, and an attribute name a, let $s{:}e$ (resp. $s{:}a$) denote the pair (s, e) (resp. (s, a)). The notation helps to clarify the role of namespaces as a means of providing contextual or scoping information to an element/attribute.

There are several consistency constraints on the node labellings.

- For each $s \in \mathrm{NS}$, there are *no* two nodes in the tree with the same value of $s{:}\texttt{id}$.
- A node cannot be labelled by both `:link` and `:visited`.

[3]See the notes at the end of the section.

- A node cannot be labelled by both `:enabled` and `:disabled`.
- Only one node in the tree may be labelled `:target`.
- A node contains the label `:root` iff it is the root node.
- A node labelled `:empty` must have no children.

In the sequel, we will tacitly assume that document trees satisfy these consistency constraints. We write $\mathrm{Trees}(\mathrm{NS}, E, A, \Gamma)$ for the set of such trees.

5.2 Definition of CSS3 selectors

In the following sections we define CSS selectors syntax and semantics. Informally, a CSS selector consists of *node selectors* σ — which match individual nodes in the tree — combined using the operators >>, >, +, and ~. These operators express the descendant-of, child-of, neighbour-of, and sibling-of relations respectively. Note that the blank space character is used instead of >> in CSS3, though we opt for the latter in the formalisation for the sake of readability. So, for example, we use `.journal` >> `.science` (i.e. choose all nodes with class `.science` that is a descendant of nodes with class `.journal`) instead of the standard syntax `.journal .science`. In addition, in order to distinguish syntax from meaning, we use slightly different notation to their counterpart semantical operators $\gg$, $>$, $+$, and $\sim$.

We remark that a comma (`,`) is not an operator in the CSS selector syntax. Instead a *selector group* is a comma-separated list of selectors that is matched if any of its selectors is matched. A CSS rule thus consists of a selector group and a list of property declarations. For the purposes of refactoring it is desirable to treat selectors individually as it allows the most flexibility in reorganising the CSS file. Hence we treat a selector group simply as a set of selectors that can be separated if needed.

A node selector σ has the form $\tau\Theta$ where τ constrains the *type* of the node. That is, τ places restrictions on the element label of the node, e.g., `p` for paragraph elements and $\star$ (or an empty string) for all elements. The rest of the selector is a set Θ of *simple* selectors (written as a concatenation of strings representing these simple selectors) that assert atomic properties of the node. There are four types of simple selectors.

Type 1: attribute selectors of the form `[`$s \mid a$ op v`]` for some namespace s, attribute a, operator op $\in \{=, \tilde{}=, |=, \hat{}=, \$=, \star=\}$, and some string $v \in \Gamma^*$. We may write `[`a op v`]` to mean that a node can be matched by `[`$s \mid a$ op v`]` for some s. The operators $=$, $\hat{}=$, $\$=$, and $\star=$ take their meaning from regular expressions. That is, equals, begins-with, ends-with, and contains respectively. The remaining operators are more subtle. The $\tilde{}=$ operator means the attribute is a string of space-separated values, one of which is v. The $|=$ operator is intended for use with language identification, e.g., as in the attribute selector `[lang |= "en-GB"]` to mean “English” as spoken in Great Britain. Thus $|=$ asserts that either the attribute has value v or is a string of the form v-v' where - is the dash character, and v' is some string. Note that if the `lang` attribute value of a node is `en-GB`, the node also matches the simple selector `[lang |= "en"]`. In addition, recall that `class` and `id` are two special attribute names. For this reason, CSS introduces the shorthands `.`v and `#`v for, respectively, the simple selectors `[class ~= `v`]` and `[id = `v`]`, i.e., asserting that the node has a class or ID v. An example of a valid CSS selector is the selector `h1.fruit.veg`, which chooses all nodes with class `fruit` and `veg`, and element name `h1` (which includes the following two elements: `<h1 class="fruit veg">` and `<h1 class="veg fruit">`).

Type 2. attribute selectors of the form `[`$s \mid a$`]`, asserting that the attribute is merely defined on the node. As before, we may write `[`a`]` to mean that the node may be matched by `[`$s \mid a$`]` for some namespace s. As an example, `img[alt]` chooses all `img` elements where the attribute `alt` is defined.

Type 3. pseudo-class label of a node, e.g., the selector `:enabled` ensures the node is currently enabled in the document. There are several further kinds of pseudo-classes that assert counting constraints on the children of a selected node. Of particular interest are selectors such as `:nth-child(`α`n + `β`)`, which assert that the node has a particular position in the sibling order. For example, `:nth-child(2n + 1)` means there is some $n \geq 0$ such that the node is the $(2n+1)$st node in the sibling order.

Type 4. negations `:not(`θ`)` of a simple selector θ with the condition that negations cannot be nested or apply to multiple atoms. For example, `:not(.fruit):not(.veg)` is a valid selector, whereas `:not(:not(.veg))` and `:not(.fruit.veg)` are *not* a valid selectors.

5.2.1 Syntax.

Fix the sets NS, E, A, and Γ. We define SEL for the set of *(CSS) selectors* and NSEL for the set of *node selectors*. The set SEL is the set of formulas φ defined as:

$$\varphi ::= \sigma \;\Big|\; \varphi \gg \sigma \;\Big|\; \varphi > \sigma \;\Big|\; \varphi + \sigma \;\Big|\; \varphi \sim \sigma$$

where $\sigma \in$ NSEL is a *node selector* with syntax $\sigma ::= \tau\Theta$ with τ having the form

$$\tau ::= * \;\Big|\; (s\,|\,*) \;\Big|\; e \;\Big|\; (s\,|\,e)$$

where $s \in$ NS and $e \in E$ and Θ is a (possibly empty) set of conditions θ with syntax

$$\theta ::= \theta_\neg \;\Big|\; \texttt{:not(}\sigma_\neg\texttt{)}$$

where $\theta_\neg$ and $\sigma_\neg$ are conditions that do not contain negations, i.e.:

$$\begin{array}{lcl}
\sigma_\neg & ::= & * \;\Big|\; (s\,|\,*) \;\Big|\; e \;\Big|\; (s\,|\,e) \;\Big|\; \theta_\neg \\
\theta_\neg & ::= & [s\,|\,a] \;\Big|\; [s\,|\,a \text{ op } v] \;\Big|\; [a] \;\Big|\; [a \text{ op } v] \;\Big| \\
& & \texttt{:link} \;\Big|\; \texttt{:visited} \;\Big|\; \texttt{:hover} \;\Big|\; \texttt{:active} \;\Big|\; \texttt{:focus} \;\Big| \\
& & \texttt{:enabled} \;\Big|\; \texttt{:disabled} \;\Big|\; \texttt{:checked} \;\Big| \\
& & \texttt{:root} \;\Big|\; \texttt{:empty} \;\Big|\; \texttt{:target} \;\Big| \\
& & \texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)} \;\Big|\; \texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)} \;\Big| \\
& & \texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)} \;\Big|\; \texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)} \\
& & \texttt{:only-child} \;\Big|\; \texttt{:only-of-type} \\
\text{op} & ::= & = \;\Big|\; \sim= \;\Big|\; |= \;\Big|\; \hat{}= \;\Big|\; \$= \;\Big|\; *=
\end{array}$$

where $s \in$ NS, $e \in E$, $a \in A$, $v \in \Gamma^*$, and $\alpha, \beta \in \mathbb{Z}$. In the sequel, whenever Θ is the empty set, we will denote the node selector $\tau\Theta$ as τ instead of $\tau\emptyset$.

5.2.2 Semantics.

The semantics of a selector is defined with respect to a document tree and a node in the tree. More precisely, the semantics of CSS3 selectors φ are defined inductively with respect to a document tree $T = (D, \lambda)$ and a node $\eta \in D$ as follows. [Note: (1) p ranges over the set P of pseudo-classes, (2) vv' is the concatenation of the strings v and v', and (3) v-v' is the concatenation of v and v' with a "-" in between.]

$$\begin{array}{lcl}
T, \eta \models \varphi \gg \sigma & \overset{\text{def}}{\Leftrightarrow} & \exists \eta' \gg \eta\,.\,(T, \eta' \models \varphi) \text{ and } (T, \eta \models \sigma) \\
T, \eta \models \varphi > \sigma & \overset{\text{def}}{\Leftrightarrow} & \exists \eta' > \eta\,.\,(T, \eta' \models \varphi) \text{ and } (T, \eta \models \sigma) \\
T, \eta \models \varphi + \sigma & \overset{\text{def}}{\Leftrightarrow} & \exists \eta' + \eta\,.\,(T, \eta' \models \varphi) \text{ and } (T, \eta \models \sigma)
\end{array}$$

$$
\begin{array}{lcl}
T,\eta \models \varphi \sim \sigma & \overset{\text{def}}{\Leftrightarrow} & \exists \eta' \sim \eta \,.\, (T,\eta' \models \varphi) \text{ and } (T,\eta \models \sigma) \\
T,\eta \models \tau\Theta & \overset{\text{def}}{\Leftrightarrow} & (T,\eta \models \tau) \text{ and } \forall \theta \in \Theta \,.\, (T,\eta \models \theta) \\
T,\eta \models \texttt{(}s\,\texttt{|}\,\texttt{*}\texttt{)} & \overset{\text{def}}{\Leftrightarrow} & s = \lambda_{\texttt{S}}(\eta) \\
T,\eta \models \texttt{*} & \overset{\text{def}}{\Leftrightarrow} & \top \\
T,\eta \models \texttt{(}s\,\texttt{|}\,e\texttt{)} & \overset{\text{def}}{\Leftrightarrow} & s = \lambda_{\texttt{S}}(\eta) \wedge e = \lambda_{\texttt{E}}(\eta) \\
T,\eta \models e & \overset{\text{def}}{\Leftrightarrow} & T,\eta \models \texttt{(}s\,\texttt{|}\,e\texttt{)} \quad \text{for some } s \in \mathrm{NS} \\
T,\eta \models p & \overset{\text{def}}{\Leftrightarrow} & p \in \lambda_{\texttt{P}}(\eta) \\
T,\eta \models \texttt{:not(}\theta_{\neg}\texttt{)} & \overset{\text{def}}{\Leftrightarrow} & \neg\,(T,\eta \models \theta_{\neg}) \\
T,\eta \models \texttt{[}a\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & T,\eta \models \texttt{[}s\,\texttt{|}\,a\texttt{]} \quad \text{for some } s \in \mathrm{NS} \\
T,\eta \models \texttt{[}a \text{ op } v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & T,\eta \models \texttt{[}s\,\texttt{|}\,a \text{ op } v\texttt{]} \quad \text{for some } s \in \mathrm{NS} \\
T,\eta \models \texttt{[}s\,\texttt{|}\,a\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & \lambda_{\texttt{A}}(\eta)(s,a) \neq \bot \\
T,\eta \models \texttt{[}s\,\texttt{|}\,a \texttt{=} v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & \lambda_{\texttt{A}}(\eta)(s,a) = v \\
T,\eta \models \texttt{[}s\,\texttt{|}\,a \texttt{ |=} v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & \left(\begin{array}{c} (\lambda_{\texttt{A}}(\eta)(s,a) = v) \text{ or} \\ \exists v' \,.\, (\lambda_{\texttt{A}}(\eta)(s,a) = v\text{-}v') \end{array} \right) \\
T,\eta \models \texttt{[}s\,\texttt{|}\,a \texttt{ ^=} v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & \exists v' \in \Gamma^* \,.\, \lambda_{\texttt{A}}(\eta)(s,a) = vv' \\
T,\eta \models \texttt{[}s\,\texttt{|}\,a \texttt{ \$=} v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & \exists v' \in \Gamma^* \,.\, \lambda_{\texttt{A}}(\eta)(s,a) = v'v \\
T,\eta \models \texttt{[}s\,\texttt{|}\,a \texttt{ *=} v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & \exists v_1, v_2 \in \Gamma^* \,.\, \lambda_{\texttt{A}}(\eta)(s,a) = v_1 v v_2
\end{array}
$$

with the missing attribute selector being (noting $v{\llcorner\!\lrcorner}v'$ is the concatenation of v and v' with the space character ${\llcorner\!\lrcorner}$ in between)

$$
\begin{array}{lcl}
T,\eta \models \texttt{[}a \texttt{ ~=} v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} & \lambda_{\texttt{A}}(\eta)(s,a) = v \text{ or } \exists v' \,.\, (\lambda_{\texttt{A}}(\eta)(s,a) = v{\llcorner\!\lrcorner}v') \text{ or} \\
 & & \exists v' \,.\, (\lambda_{\texttt{A}}(\eta)(s,a) = v'{\llcorner\!\lrcorner}v) \text{ or } \exists v_1, v_2 \,.\, (\lambda_{\texttt{A}}(\eta)(s,a) = v_1{\llcorner\!\lrcorner}v{\llcorner\!\lrcorner}v_2)
\end{array}
$$

then, for the counting selectors

$$
\begin{array}{rcl}
T,\eta \models \texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)} & \overset{\text{def}}{\Leftrightarrow} & \text{there is some } n \in \mathbb{Z}_{>0} \text{ such that } \eta \text{ is the } \alpha n + \beta\text{th child} \\
T,\eta \models \texttt{:only-child} & \overset{\text{def}}{\Leftrightarrow} & \text{the parent of } \eta \text{ has precisely one child} \\
T,\eta \models \texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)} & \overset{\text{def}}{\Leftrightarrow} & \text{there is some } n \in \mathbb{Z}_{>0} \text{ such that the parent of } \eta \text{ has precisely } \alpha n + \beta - 1 \text{ children with namespace } \lambda_{\texttt{S}}(\eta) \text{ and element name } \lambda_{\texttt{E}}(\eta) \text{ for some } n \text{ that are (strictly) preceding siblings of } \eta \\
T,\eta \models \texttt{:only-of-type} & \overset{\text{def}}{\Leftrightarrow} & \text{the parent of } \eta \text{ has precisely one child with}
\end{array}
$$

namespace $\lambda_{\mathrm{S}}(\eta)$ and element name $\lambda_{\mathrm{E}}(\eta)$

Finally, the semantics of the remaining two selectors, which are `:nth-last-child(`α`n + `β`)` and `:nth-last-of-type(`α`n + `β`)`, is exactly the same as `:nth-child(`α`n + `β`)` and `:nth-of-type(`α`n + `β`)`, respectively, except with the sibling ordering reversed (i.e. the right-most child of a parent is treated as the first).

REMARK 5.1. *Readers familiar with HTML may have expected more constraints in the semantics. For example, if a node matches* `:hover`*, then its parent should also match* `:hover`*. However, this is part of the HTML5 specification, not of CSS3. In fact, the CSS3 selectors specification explicitly states that a node matching* `:hover` *does not imply its parent must also match* `:hover`.

5.2.3 Divergences from full CSS. Note that we diverge from the full CSS specification in a number of places. However, we do not lose expressivity.

- We assume each element has a namespace. In particular, we do not allow elements without a namespace. There is no loss of generality here since we can simply assume a "null" namespace is used instead. Moreover, we do not support default name spaces and assume namespaces are explicitly given.
- We did not include `:lang (`l`)`. Instead, we will assume (for convenience) that all nodes are labelled with a language attribute with some fixed namespace s. In this case, `:lang (`l`)` is equivalent[4] to `[`s`|lang |=`l`]`.
- We did not include `:indeterminate` since it is not formally part of the CSS3 specification.
- We omit the selectors `:first-child` and `:last-child`, as well as `:first-of-type` and `:last-of-type`, since they are expressible using the other operators.
- We omitted `even` and `odd` from the nth child operators since these are easily definable as $2n$ and $2n+1$.
- We do not explicitly handle document fragments. These may be handled in a number of ways. For example, by adding a phantom root element (since the root of a document fragment does not match `:root`) with a fresh ID ι and adjusting each node selector in the CSS selector to assert `:not(#`ι`)`. Similarly, lists of document fragments can be modelled by adding several subtrees to the phantom root.
- A CSS selector can be suffixed with a *pseudo-element* of the form `::first-line`, `::first-letter`, `::before`, and `::after`. Pseudo-elements are easy to handle and only provide a distraction to our presentation. For this reason, we relegate them into the appendix.
- We define our DOM trees to use a finite alphabet Γ. Currently the CSS3 selectors specification uses Unicode as its alphabet for lexing. Although the CSS3 specification is not explicit about the finiteness of characters appearing in potential DOMs, since Unicode is finite (Unicode, Inc. 2016) (with a maximal possible codepoint) we feel it is reasonable to assume DOMs are also defined over a finite alphabet.

5.3 Solving the intersection problem

We now address the problem of checking the intersection of two CSS selectors. Let us write

$$[\![\varphi]\!] := \{(T,\eta) : T,\eta \models \varphi\}$$

[4] The CSS specification defines `:lang (`l`)` in this way. A restriction of the language values to standardised language codes is only a recommendation.

to denote the set of pairs of tree and node satisfying the selector φ. The *intersection problem of CSS selectors* is to decide if $[\![\varphi]\!] \cap [\![\varphi']\!] \neq \emptyset$, for two given selectors φ and φ'. A closely related decision problem is the *non-emptiness problem of CSS selectors*, which is to decide if $[\![\varphi]\!] \neq \emptyset$, for a given selector φ. The two problems are *not* the same since CSS selectors are not closed under intersection (i.e. the conjunction of two CSS selectors is in general not a valid CSS selector).

THEOREM 5.1 (NON-EMPTINESS). *The non-emptiness problem for CSS selectors is efficiently reducible to satisfiability over quantifier-free theory over integer linear arithmetic. Moreover, the problem is is NP-complete.*

THEOREM 5.2 (INTERSECTION). *The intersection problem for CSS selectors is efficiently reducible to satisfiability over quantifier-free theory over integer linear arithmetic. Moreover, the problem is NP-complete.*

Recall from Section 3 that satisfiability over quantifier-free theory over integer linear arithmetic is in NP and can be solved by a highly-optimised SMT solver (e.g. Z3 (de Moura and Bjørner 2008)). The NP-hardness in the above theorems suggests that our SMT-based approach is theoretically optimal. In addition, our experiments with real-world selectors (see Section 8) suggest that our SMT-based approach is fast in practice, with each problem instance solved within instants.

Idea behind our SMT-based approach. We now provide the idea behind the efficient reduction to quantifier-free theory over integer linear arithmetic. Our reduction first goes via a new class of tree automata (called CSS automata), which — like CSS selectors — are symbolic representations of sets of pairs containing a document tree and a node in the tree. We will call such sets *languages recognised by the automata*. Given a CSS selector φ, we can efficiently construct a CSS automaton $\mathcal{A}$ that can symbolically represent $[\![\varphi]\!]$. Unlike CSS selectors, however, we will see that languages recognised by CSS automata enjoy closure under intersection, which will allow us to treat the intersection problem as the non-emptiness problem. More precisely, a CSS automaton navigates a tree structure in a similar manner to a CSS selector: transitions may only move down the tree or to a sibling, while checking a number of properties on the visited nodes. The difficulty of taking a direct intersection of two selectors is that the two selectors may descend to different child nodes, and then meet again after the application of a number of sibling combinators, i.e., their paths may diverge and combine several times. CSS automata overcome this difficulty by always descending to the *first* child, and then move from *sibling to sibling*. Thus, the intersection of CSS automata can be done with a straightforward automata product construction, e.g., see (Vardi 1995).

Next, we show that the non-emptiness of CSS automata can be decided in NP by a polynomial-time reduction to satisfiability of quantifier-free theory of integer linear arithmetic. Ideally, we would like to show that if a CSS automaton has a non-empty language, then it accepts a small tree (i.e. with polynomially many nodes). This is unfortunately not the case, as the reader can see in our NP-hardness proof idea below. Therefore, we use a different strategy. First , we prove three "small model lemmas". The first is quite straightforward and shows that, to prove non-emptiness, it suffices to consider a witnessing automata run of length n for an automaton with n transitions (each automata transition allows some nodes to be skipped). Secondly, we show that it suffices to consider attribute selector values (i.e. strings) of length linear in the size of the CSS automata. This is non-trivial and uses a construction inspired by (Muscholl and Walukiewicz 2005). Thirdly, we show that it suffices to consider trees whose sets of namespaces and element names are linear in the size of the CSS automaton. Our formula φ attempts to guess this automata run, the attribute selector values, element names, and namespaces. The global ID constraint (i.e. all the guessed IDs are distinct) can be easily asserted in the formula. So far, boolean variables are sufficient owing to the small model lemmas (which allow us to do bit-blasting). *Where, then, do the integer variables come into* *play?* For each position i in the guessed path, we introduce an integer variable $\overline{n}_i$ to denote that the node at position i in the path is the $\overline{n}_i$th child. This is necessary if we want to assert counting constraint like `:nth-child(`α`n + `β`)`, which would be encoded in integer linear arithmetic as $\exists \overline{n} : \overline{n}_i = \alpha\overline{n} + \beta$.

Proof Idea of NP-hardness. We now provide an intuition on how to prove NP-hardness in the above theorems. First, observe that the intersection is computationally at least as hard as the non-emptiness problem since we can set the second selector to be $\star$. To prove NP-hardness of the non-emptiness is by a polynomial-time reduction from the NP-complete problem of *non-universality of unions of arithmetic progressions* (Stockmeyer and Meyer 1973, Proof of Theorem 6.1). Examples of arithmetic progressions are $2\mathbb{N}+1$ and $5\mathbb{N}+2$, which are shorthands for the sets $\{1,3,5,\ldots\}$ and $\{2,7,12,\ldots\}$, respectively. The aforementioned non-universality problem allows an arbitrary number of arithmetic progressions as part of the input and we are interested in checking whether the union equals the entire set $\mathbb{N}$ of natural numbers. As an example of the reduction, checking $\mathbb{N} \neq 2\mathbb{N}+1 \cup 5\mathbb{N}+2$ is equivalent to the non-emptiness of

```
:not(root):not(:nth-child(2n+2)):not(:nth-child(5n+3))
```

which can be paraphrased as checking the existence of a tree with a node that is neither the root, nor the $2n+2$nd child, nor the $5n+3$rd child. Observe that we add 1 to the offset of the arithmetic progressions since the selector `:nth-child` starts counting (the number of children) from 1, not from 0. A full NP-hardness proof is available in Appendix B.2.

5.4 Extracting the edge order $\prec$ from a CSS file

Recall that our original goal is to compute a CSS-graph $\mathcal{G} = (S, P, E, \prec, \mathtt{wt})$ from a given CSS file. The sets P, S, and E, and the function `wt` can be computed easily as explained in Section 4. We now show how to compute $\prec$ using the algorithm for checking intersection of two selectors. We present an intuitive ordering, before explaining how this may be relaxed while still preserving the semantics.

An initial definition of $\prec$ is simple to obtain: we want to order $(s,p) \prec (s',p')$ whenever (s',p') appears later in the CSS file than (s,p), the selectors may overlap but are not distinguished by their specificity, and p and p' assign conflicting values to a given property name. More formally, we first compute the specificity of all the selectors in the CSS file. This can be easily computed in the standard way (Çelik et al. 2011). Now, the relation $\prec$ can only relate two edges $(s,p),(s',p') \in E$ satisfying

(1) s and s' have the same specificity,
(2) we have $p \neq p'$ but the property names for p and p' are the same (e.g. $p =$ `color:blue` and $p' =$ `color:red` with property name `color`), and
(3) s intersects with s' (i.e. $[\![s]\!] \cap [\![s']\!] \neq \emptyset$).

If both (1) and (2) are satisfied, Condition (3) can be checked by means of SMT-solver via the reduction in Theorem 5.2. Supposing that Condition (3) holds, we simply compute the indices of the edges in the file: $m := \mathrm{index}((s,p))$ and $m' := \mathrm{index}((s',p'))$. We put $(s,p) \prec (s',p')$ iff $m < m'$. [There are two minor technical details with the keyword `!important` and *shorthand property names*. See Appendix B.3.]

The ordering given above unfortunately turns out to be too conservative. In the following, we give an example to demonstrate this, and propose a refinement to the ordering. Consider the CSS file

```
.a { color:red; color:rgba(255,0,0,0.5) }
.b { color:red; color:rgba(255,0,0,0.5) }
```

In this file, both nodes matching `.a` and `.b` are assigned a semi-transparent red with solid red being defined as a *fallback* when the semi-transparent red is not supported. If the edge order is calculated as above, we obtain

$$(\texttt{.a}, \texttt{color:rgba(255,0,0,0.5)}) \lessdot (\texttt{.b}, \texttt{color:red}) \tag{1}$$

which prevents the obvious refactoring

```
.a, .b { color:red; color:rgba(255,0,0,0.5) }
```

The key observation is that the fact that we also have

$$(\texttt{.b}, \texttt{color:red}) \lessdot (\texttt{.b}, \texttt{color:rgba(255,0,0,0.5)}) \tag{2}$$

renders any violations of (1) benign: such a violation would give precedence to the declaration `color:rgba(255,0,0,0.5)` over `color:red` for nodes matching both `.a` and `.b`. However, because of (2) this should happen anyway and we can omit (1) from $\lessdot$.

Formally, the ordering we need is as follows. If Conditions (1-3) hold, we compute $m := \text{index}((s,p))$ and $m' := \text{index}((s',p'))$ and put $(s,p) \lessdot (s',p')$ iff

- $m < m'$, and
- (s',p) does not exist or $\text{index}((s',p)) < m'$.

That is, we omit $(s,p) \lessdot (s',p')$ if (s',p) appears later in the CSS file (i.e. $\text{index}((s',p')) < \text{index}((s',p))$). Note, we are guaranteed in this latter case to include $(s',p') \lessdot (s',p)$ since (s',p') and (s',p) can easily be seen to satisfy the conditions for introducing $(s',p') \lessdot (s',p)$.

6 MORE DETAILS ON SOLVING SELECTOR INTERSECTION PROBLEM

In the previous section, we have given the intuition behind the efficient reduction from the CSS selector intersection problem to quantifier-free theory over integer linear arithmetic, for which there is a highly-optimised SMT-solver (de Moura and Bjørner 2008). In this section, we present this reduction in full, which may be skipped on a first reading without affecting the flow of the paper.

This section is structured as follows. We begin by defining CSS automata, and show their closure under intersection. The latter reduces the intersection problem of CSS automata to the non-emptiness problem of CSS automata. We then provide a semantic-preserving transformation from CSS selectors to CSS automata. Finally, we provide a reduction from non-emptiness of CSS automata to satisfiability over quantifier-free integer linear arithmetic. We will see that each such transformation/reduction runs in polynomial-time, resulting in the complexity upper bound of NP, which is precise due to the NP-hardness of the problem from the previous section.

6.1 CSS Automata

CSS automata are a kind of finite automata which navigate the tree structure of a document tree. Transitions of the automata will contain one of four labels: $\downarrow$, $\rightarrow$, $\rightarrow_+$, and $\circ$. Intuitively, these transitions perform the following operations. $\downarrow$ moves to the first child of the current node. $\rightarrow$ moves to the next sibling of the current node. $\rightarrow_+$ moves an arbitrary number of siblings to the right. Finally, $\circ$ reads the node matched by the automaton. Since CSS does not have loops, we require only self loops in our automata, which are used to skip over nodes (e.g. $.v$ ~ $.v'$ may pass over many nodes between that matching $.v$ and that matching $.v'$). We do not allow $\rightarrow$ to label a loop – this is for the purposes of the NP proof: it can be more usefully represented as $\rightarrow_+$.

An astute reader may complain that $\rightarrow_+$ does not need to appear on a loop since it can already pass over an arbitrary number of nodes. However, the product construction used for intersection becomes easier if $\rightarrow_+$ appears only on loops. There is no analogue of $\rightarrow_+$ for $\downarrow$ because we do not need it: the use of $\rightarrow_+$ is motivated by selectors such as `:nth-child(`α`n + `β`)` which count the number of siblings of a node. No CSS selector counts the number of descendants/ancestors.

Formal Definition of CSS Automata. A *CSS Automaton* $\mathcal{A}$ is a tuple $\left(Q, \Delta, q^{\text{in}}, q_f\right)$ where Q is a finite set of states, $\Delta \subseteq Q \times \{\downarrow, \rightarrow, \rightarrow_+, \circ\} \times \text{NSEL} \times Q$ is a transition relation, $q^{\text{in}} \in Q$ is the initial state, and $q_f \in Q$ is the final state. Moreover,

(1) (only self-loops) there exists a partial order $\lesssim$ such that $(q, d, \sigma, q') \in \Delta$ implies $q' \lesssim q$,
(2) ($\rightarrow_+$ loops and doesn't check nodes) for all $(q, \rightarrow_+, \sigma, q') \in \Delta$ we have $q = q'$ and $\sigma = \star$, and
(3) ($\rightarrow$ doesn't label loops) for all $(q, d, \sigma, q) \in \Delta$ we have $d \neq \rightarrow$ and $\sigma = \star$.
(4) ($\circ$ checks last node only) for all $(q, d, \sigma, q') \in \Delta$ we have $q' = q_f$ iff $d = \circ$.
(5) (q_f is a sink) for all $(q, d, \sigma, q') \in \Delta$ we have $q \neq q_f$.

We now define the semantics of CSS automata, i.e., given an automaton $\mathcal{A}$, which language $\mathcal{L}(\mathcal{A})$ they recognise. Intuitively, the set $\mathcal{L}(\mathcal{A})$ contains the set of pairs of document tree and node, which the automaton $\mathcal{A}$ accepts. We will now define this more formally. Write $q \xrightarrow[\sigma]{d} q'$ to denote a transition $(q, d, \sigma, q') \in \Delta$. A document tree $T = (D, \lambda)$ and node $\eta \in D$ is *accepted* by a CSS automaton $\mathcal{A}$ if there exists a sequence

$$q_0, \eta_0, q_1, \eta_1, \ldots, q_\ell, \eta_\ell, q_{\ell+1} \in (Q \times D)^* \times \left\{q_f\right\}$$

such that $q_0 = q^{\text{in}}$ is the initial state, $\eta_0 = \varepsilon$ is the root node, $q_{\ell+1} = q_f$ is the final state, $\eta_\ell = \eta$ is the matched node, and for all i, there is some transition $q_i \xrightarrow[\sigma]{d} q_{i+1}$ with η_i satisfying σ and if $i \leq \ell$,

(1) if $d = \downarrow$ then $\eta_{i+1} = \eta_i 1$, (i.e., the leftmost child of η_i)
(2) if $d = \rightarrow$ then there is some η' and ι such that $\eta_i = \eta' \iota$ and $\eta_{i+1} = \eta'(\iota + 1)$, and
(3) if $d = \rightarrow_+$ then there is some η', ι and ι' such that $\eta_i = \eta' \iota$ and $\eta_{i+1} = \eta' \iota'$ and $\iota' > \iota$.

Such a sequence is called an *accepting run* of length ℓ. The *language $\mathcal{L}(\mathcal{A})$ recognised by $\mathcal{A}$* is the set of pairs (T, η) accepted by $\mathcal{A}$.

6.2 Transforming CSS Selectors to CSS Automata

The following proposition shows that CSS automata is no less expressive than CSS selectors.

PROPOSITION 6.1. *For each CSS selector φ, we may construct in polynomial-time a CSS automaton $\mathcal{A}_\varphi$ such that $\mathcal{L}(\mathcal{A}_\varphi) = [\![\varphi]\!]$.*

We show this proposition by giving a translation from a given CSS selector to a CSS automaton. Before the formal definition, we consider the a simple example. A more complex example is shown after the translation.

6.2.1 *Simple Example.*

Consider the selector

```
p + .a
```

which selects a node that has a class `a` and is directly a right neighbour of a node with element `p`. Figure 3 gives a CSS automaton representing the selector. The automaton begins with a loop that can navigate down any branch of the tree using the $\downarrow$ and $\rightarrow_+$ transitions from $\circ_1$. Then, since it always moves from the first child to the last, it will first see the node with the `p`. When reading this node, it will move to the next child using $\rightarrow$ before matching the node with class `a`, leading to the accepting state.

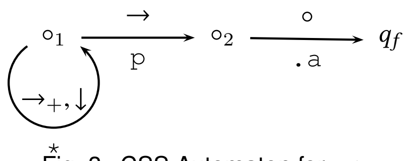
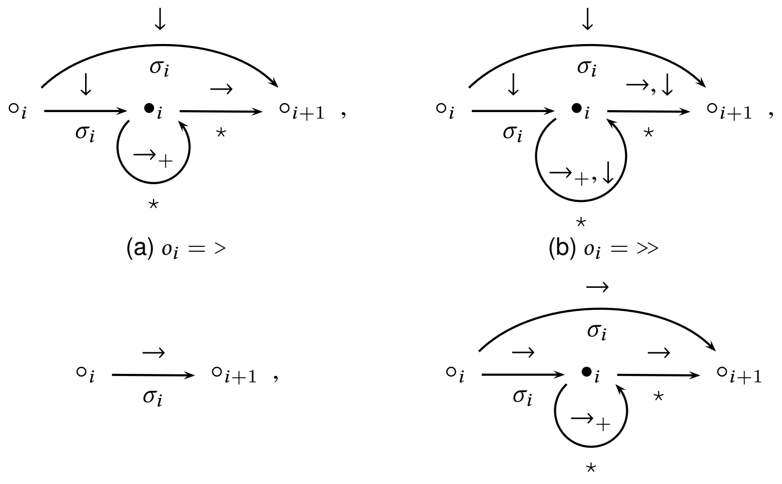


Fig. 3. CSS Automaton for `p + .a`.



(a) $o_i = >$ (b) $o_i = >>$

(c) $o_i = +$ (d) $o_i = \;\tilde{}$

Fig. 4. Converting selectors to CSS automata.

6.2.2 Formal Translation. Given a CSS selector φ, we define $\mathcal{A}_\varphi$ as follows. We can write φ uniquely in the form

$$\sigma_1 \; o_1 \; \sigma_2 \; o_2 \; \cdots \; o_{n-1} \; \sigma_n$$

where each σ_i is a node selector, and each $o_i \in \{>>, >, +, \tilde{}\}$. We will have a state $\circ_i$ corresponding to each σ_i, o_i. We define

$$\mathcal{A}_\varphi = \left(Q, E, \Delta, \circ_1, q_f\right)$$

where $Q = \{\circ_i, \bullet_i \mid 1 \leq i \leq n\} \uplus \left\{q_f\right\}$ and we define the transition relation Δ to contain the following transitions. The initial and final transitions are used to navigate from the root of the tree to the node matched by σ_1, and to read the final node matched by σ_n (and the selector as a whole) respectively. That is, $\circ_1 \xrightarrow[\star]{\downarrow,\rightarrow_+} \circ_1$ and $\circ_n \xrightarrow[\sigma_n]{\circ} q_f$. We have further transitions for $1 \leq i < n$ that are shown in Figure 4. The transitions connect $\circ_i$ to $\circ_{i+1}$ depending on o_i. Figure 4a shows the child operator. The automaton moves to the first child of the current node and move to the right zero or more steps. Figure 4b shows the descendant operator. The automaton traverses the tree downward and rightward any number of steps. The neighbour operator is handled in Figure 4c by simply moving to the next sibling. Finally, the sibling operator is shown in Figure 4d.

We prove the correctness of this construction in Lemma C.1 (soundness) and Lemma C.2 (completeness) in Appendix C.1.

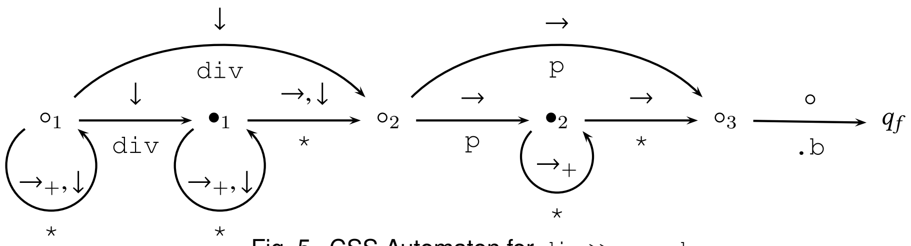


Fig. 5. CSS Automaton for `div >> p ~ .b`

6.2.3 Complex Example. Figure 5 gives an example of a CSS automaton representing the more complex selector

`div >> p ~ .b`

which selects a node that has class `b`, is a right sibling of a node with element `p` and moreover is a descendent of a `div` node. This automaton again begins at $\circ_1$ and navigates until it finds the node with the `div` element name. The automaton can read this node in two ways. The topmost transition covers the case where the `p` node is directly below the `div` node. The lower transition allows the automaton to match the `p` node and then use a loop to navigate to the descendent node that will match `p`. Similarly, from $\circ_2$ the automaton can read a `p` node and choose between immediately matching the node with class `b` or navigating across several siblings (using the loop at state $\bullet_2$) before matching `.b` and accepting.

6.3 Closure Under Intersection

The problems of non-emptiness and intersection of CSS automata can be defined in precisely the same way we defined them for CSS selectors. One key property of CSS automata, which is not enjoyed by CSS selectors, is the closure of their languages under intersection. This allows us to treat the problem of intersection of CSS automata (i.e. the non-emptiness of the intersection of two CSS automata languages) as the non-emptiness problem (i.e. whether a given CSS automaton has an empty language).

PROPOSITION 6.2. *Given two CSS automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we may construct in polynomial-time an automaton $\mathcal{A}_1 \cap \mathcal{A}_2$ such that $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2)$.*

The construction of the CSS automaton $\mathcal{A}_1 \cap \mathcal{A}_2$ is by a variant of the standard product construction (Vardi 1995) for finite-state automata over finite words, which run the two given automata in parallel synchronised by the input word. Our construction runs the two CSS automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in parallel synchronised by the path that they traverse. We first proceed with the formal definition and give an example afterwards.

6.3.1 Formal Definition of Intersection. We first define the intersection of two node selectors. Recall node selectors are of the form $\tau\Theta$ where $\tau \in \{\star, (s \mid \star), e, (s \mid e) \mid s \in \mathrm{NS} \wedge e \in E\}$. The intersection of two node selectors $\tau_1\Theta_1$ and $\tau_2\Theta_2$ should enforce all properties defined in Θ_1 and Θ_2. In addition, both selectors should be able to agree on the namespace and element name of the node, hence this part of the selector needs to be combined more carefully. Thus, letting $\Theta = \Theta_1 \cup \Theta_2$.

we define

$$\tau_1\Theta_1 \cap \tau_2\Theta_2 = \begin{cases} \tau_2\Theta & \tau_1 = \star \\ \tau_1\Theta & \tau_2 = \star \\ \tau_2\Theta & \tau_1 = (s \mid \star) \wedge \begin{pmatrix} \tau_2 = (s \mid \star) \vee \\ \tau_2 = (s \mid e) \end{pmatrix} \\ (s \mid e)\,\Theta & \tau_1 = (s \mid \star) \wedge \tau_2 = e \\ \tau_1\Theta & \tau_1 = (s \mid e) \wedge \begin{pmatrix} \tau_2 = (s \mid e) \vee \\ \tau_2 = e \vee \\ \tau_2 = (s \mid \star) \end{pmatrix} \\ \tau_2\Theta & \tau_1 = e \wedge (\tau_2 = (s \mid e) \vee \tau_2 = e) \\ (s \mid e)\,\Theta & \tau_1 = e \wedge \tau_2 = (s \mid \star) \\ \texttt{:not}\,(\star) & \text{otherwise.} \end{cases}$$

We now define the automaton $\mathcal{A}_1 \cap \mathcal{A}_2$. The intersection automaton synchronises transitions that move in the same direction (by $\downarrow$, $\rightarrow$, $\rightarrow_+$) or both agree to match the current node at the same time (with $\circ$). In addition, we observe that a $\rightarrow_+$ can be used by one automaton while the other uses $\rightarrow$. Given

$$\mathcal{A}_1 = \left(Q_1, E, \Delta_1, q_1^{\text{in}}, q_f^1\right) \quad \text{and} \quad \mathcal{A}_2 = \left(Q_2, E, \Delta_2, q_2^{\text{in}}, q_f^2\right)$$

we define

$$\mathcal{A}_1 \cap \mathcal{A}_2 = \left(Q_1 \times Q_2, E, \Delta, \left(q_1^{\text{in}}, q_2^{\text{in}}\right), \left(q_f^1, q_f^2\right)\right)$$

where (letting d range over $\{\rightarrow, \rightarrow_+, \downarrow, \circ\}$) we set $\Delta =$

$$\begin{aligned} &\left\{(q_1, q_2) \xrightarrow[\sigma_1 \cap \sigma_2]{d} \left(q_1', q_2'\right) \,\middle|\, q_1 \xrightarrow[\sigma_1]{d} q_1' \wedge q_2 \xrightarrow[\sigma_2]{d} q_2'\right\} \cup \\ &\left\{(q_1, q_2) \xrightarrow[\sigma_1]{\rightarrow} \left(q_1', q_2\right) \,\middle|\, q_1 \xrightarrow[\sigma_1]{\rightarrow} q_1' \wedge q_2 \xrightarrow[\star]{\rightarrow_+} q_2\right\} \cup \\ &\left\{(q_1, q_2) \xrightarrow[\sigma_2]{\rightarrow} \left(q_1, q_2'\right) \,\middle|\, q_1 \xrightarrow[\star]{\rightarrow_+} q_1 \wedge q_2 \xrightarrow[\sigma_2]{\rightarrow} q_2'\right\} . \end{aligned}$$

6.3.2 Example of Intersection. Recall the automaton in Figure 3 (equivalent to `p + .a`) and the automaton in Figure 5 (equivalent to `div >> p ~ .b`). The intersection of the two automata is given in Figure 6. Each state is a tuple (q_1, q_2) where the first component q_1 represents the state of the automaton equivalent to `p + .a` and the second component q_2 the automaton equivalent to `div >> p ~ .b`.

In this example, accepting runs of the automaton will use only the top row of states. The lower states are reached when the two automata move out of sync and can no longer reach agreement on the final node matched. Usually this is by the first automaton matching a node labelled `p`, after which it must immediately accept the neighbouring node. This leaves the second automaton unable find a match. Hence, the first automaton needs to stay in state $\circ_1$ until the second has reached a near-final state. Note, the two automata need not match the same node with element name `p`.

6.4 Reducing Non-emptiness of CSS Automata to SMT-solving

We will now provide a polynomial-time reduction from the non-emptiness of a CSS automaton to satisfiability of quantifier-free theory over integer linear arithmetic. That is, given a CSS automaton $\mathcal{A}$, our algorithm constructs a quantifier-free formula $\theta_{\mathcal{A}}$ over integer linear arithmetic such that $\mathcal{A}$ recognises a non-empty language iff $\theta_{\mathcal{A}}$ is satisfiable. The encoding is quite involved and requires three small model properties discussed earlier. Once we have these properties we can construct the required formula of the quantifier-free theory over linear arithmetic. We begin by discussing each of

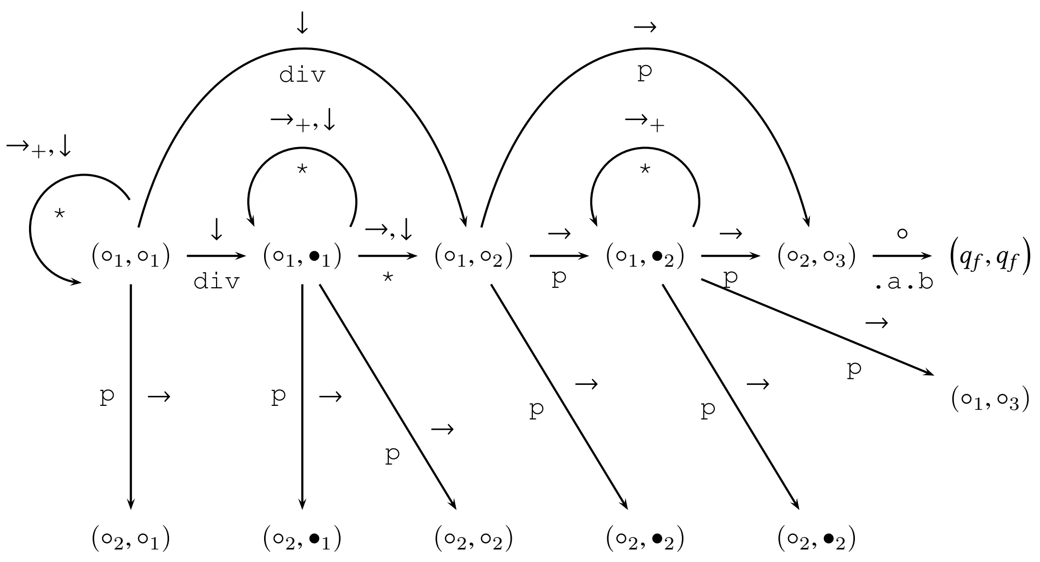


Fig. 6. The intersection of the automaton in Figure 3 (in the first component) and the automaton in Figure 5 (in the second component).

these properties in turn, and then provide the reduction. The reduction is presented in a number of stages. We show how to handle attribute selectors separately before handling the encoding of CSS automata. The encoding of CSS automata is further broken down: we first describe the variables used in the encoding, then we describe how to handle node selectors, finally we put it all together to encode runs of a CSS automaton.

For the remainder of the section, we fix a CSS automaton $\mathcal{A} = \left(Q, \Delta, q^{\text{in}}, q_f\right)$ and show how to construct the formula $\theta_{\mathcal{A}}$ in polynomial-time.

6.4.1 Bounded Run Length. The first property required is that the length of runs can be bounded. That is, if the language of $\mathcal{A}$ is not empty, there is an accepting run over some tree whose length is smaller than the derived bound. We will construct a formula that will capture all runs of length up to this bound. Thanks to the bound we know that if an accepting run exists, the formula will encode at least one, and hence be satisfiable.

PROPOSITION 6.3 (BOUNDED RUNS). *Given a CSS Automaton $\mathcal{A} = \left(Q, \Delta, q^{in}, q_f\right)$, if $\mathcal{L}(\mathcal{A}) \neq \emptyset$, there exists $(T, \eta) \in \mathcal{L}(\mathcal{A})$ with an accepting run of length $|\Delta|$.*

This proposition is straightforward to obtain. We exploit that any loop in the automaton is a self loop that only needs to be taken at most once. For loops labelled $\downarrow$, a CSS formula cannot track the depth in the tree, so repeated uses of the loop will only introduce redundant nodes. For loops labelled $\rightarrow_+$, selectors such as `:nth-child(`α`n + `β`)` may enforce the existence of a number of intermediate nodes. But, since $\rightarrow_+$ can cross several nodes, such loops also only needs to be taken once. Hence, each transition only needs to appear once in an accepting run. That is, if there is an accepting run of a CSS automaton with n transitions, there is also an accepting run of length at most n.

6.4.2 Bounding Namespaces and Elements. It will also be necessary for us to argue that the number of namespaces and elements can be bounded linearly in the size of the automaton. This is because our formula will need keep track of the number of nodes of each type appearing in the tree – this is required for encoding, e.g., the pseudo-classes of the form `:nth-of-type(`α`n + `β`)`. By bounding the number of types, our formula can use a bounded number of variables to store this information.

We state the property below. The proof is straightforward and appears in Appendix C.3.1. Intuitively, since only a finite number of nodes can be directly inspected by a CSS automaton, all others can be relabelled to a dummy type unless their type matches one of the inspected nodes.

Proposition 6.4 (Bounded Types). *Given a CSS Automaton $\mathcal{A} = \left(Q, \Delta, q^{in}, q_f\right)$ if there exists $(T, \eta) \in \mathcal{L}(\mathcal{A})$ with $T = (D, \lambda)$, then there exists some $(T', \eta) \in \mathcal{L}(\mathcal{A})$ where $T' = (D, \lambda')$ and that the image of λ' contains the set $\rfloor(E)$ of element names and and the set $\rfloor(\mathrm{NS})$ of namespaces, each of whose size is bounded linearly in the size of $\mathcal{A}$.*

6.4.3 Bounding Attribute Values. We will need to encode the satisfiability of conjunctions of attribute selectors. This is another potentially source of unboundedness because the values are strings of arbitrary length. We show that, in fact, if the language of the automaton is not empty, there there is a solution whose attribute values are strings of a length less than a bound polynomial in the size of the automaton.

The proof of the following lemma is highly non-trivial and uses techniques inspired by results in Linear Temporal Logic and automata theory. To preserve the flow of the article, we present the proof in Appendix C.3.2.

Proposition 6.5 (Bounded Attributes). *Given a CSS Automaton $\mathcal{A} = \left(Q, \Delta, q^{in}, q_f\right)$ if there exists $(T, \eta) \in \mathcal{L}(\mathcal{A})$ with $T = (D, \lambda)$, then there exists some bound N polynomial in the size of $\mathcal{A}$ and some $(T', \eta) \in \mathcal{L}(\mathcal{A})$ where the length of all attribute values in T' is bound by N.*

Given such a bound on the length of values, we can use quantifier-free Presburger formulas to "guess" these witnessing strings by using a variable for each character position in the string. Then, the letter in each position is encoded by a number. This process is discussed in the next section.

6.4.4 Encoding Attribute Selectors. Before discussing the full encoding, we first show how our formula can encode attribute selectors. Once we have this encoding, we can invoke it as a sub-routine of our main encoding whenever we have to handle attribute selectors. It is useful for readability reasons to present this in its own section.

The first key observation is that we can assume each positive attribute selector that does not specify a namespace applies to a unique, fresh, namespace. Thus, these selectors do not interact with any other positive attribute selectors and we can handle them easily. Note, these fresh namespaces do not appear in $\rfloor(\mathrm{NS})$.

We present our encoding which works by identifying combinations of attribute selectors than must apply to the same attribute value. That is, we discover how many attribute values are needed, and collect together all selectors that apply to each selector. To that end, let op range over the set $\{=, \tilde{}=, |=, \hat{}=, \$=, *=\}$ and let $\tau\Theta$ be a node selector. For each s and a, let Θ^s_a be the set of conditions in Θ of the form θ or `:not(`θ`)` where θ is of the form `[`$s \mid a$`]` or `[`$s \mid a \text{ op } v$`]`. Recall we are encoding runs of a CSS automaton of length at most n. For a given position i in the run, we define $\mathrm{AttsPres}(\tau\Theta, i)$ to be the conjunction of the following constraints, where the encoding for $\mathrm{AttsPres}_{s:a}(\Theta, i)$ is presented in the sequel. Since a constraint of the form `:not([a op v])` applies to all attributes a regardless of their namespace, we define for convenience $\mathrm{Neg}(s, a) = \{$`:not([s|a op v])` $|$ `:not([a op v])` $\in \Theta\}$.

- For each s and a with Θ_a^s non-empty and containing at least one selector of the form `[`s`|`a`]` or `[`s`|`a op v`]`, we enforce
$$\mathrm{AttsPres}_{s:a}(\Theta_a^s \cup \mathrm{Neg}(s, a), i)$$
if `:not([a])` $\notin \Theta$ and `:not([s|a])` $\notin \Theta$, else we assert false.
- For each `[`a`]` $\in \Theta$, let s be fresh namespace. We assert
$$\mathrm{AttsPres}_{s:a}(\{\,\texttt{[}s\,|\,a\texttt{]}\,\} \cup \mathrm{Neg}(s, a), i)$$
and for each `[`a op v`]` $\in \Theta$ we assert
$$\mathrm{AttsPres}_{s:a}(\{\,\texttt{[}s\,|\,a \text{ op } v\texttt{]}\,\} \cup \mathrm{Neg}(s, a), i)$$
whenever, in both cases, `:not([a])` $\notin \Theta$. If `:not([a])` $\in \Theta$ in both cases we assert false.

It remains to encode $\mathrm{AttsPres}_{s:a}(C, i)$ for some set of attribute selectors C all applying to s and a.

We can obtain a polynomially-sized global bound $(N - 1)$ on the length of any satisfying value of an attribute $s{:}a$ at some position i of the run from Proposition 6.5 (Bounded Attributes)[5]. Finally, we increment the bound by one to allow space for a trailing null character.

Once we have a bound on the length of a satisfying value, we can introduce variables $x_{i,1}^{s:a}, \ldots, x_{i,N}^{s:a}$ for each character position of the satisfying value, and encode the constraints almost directly. That is, letting θ range over positive attribute selectors, we define[6]

$$\begin{aligned}\mathrm{AttsPres}_{s:a}(C, i) \quad = \quad & \bigwedge_{\theta \in C} \mathrm{AttsPres}(\theta, \vec{x}) \ \wedge \\ & \bigwedge_{\texttt{:not(}\theta\texttt{)} \in C} \neg \mathrm{AttsPres}(\theta, \vec{x}) \ \wedge \mathrm{Nulls}(\vec{x}) \ .\end{aligned}$$

where $\vec{x} = x_{i,1}^{s:a}, \ldots, x_{i,N}^{s:a}$ will be existentially quantified later in the encoding and whose values will range[7] over $\Gamma \uplus \{0\}$ where 0 is a null character used to pad the suffix of each word. We define $\mathrm{AttsPres}(\theta, \vec{x})$ for several θ, the rest can be defined in the same way (see Appendix C.3.3). Letting $v = a_1 \ldots a_m$,

$$\begin{aligned}\mathrm{AttsPres}(\texttt{[}s\,|\,a\texttt{]}, \vec{x}) &= \top \\ \mathrm{AttsPres}(\texttt{[}s\,|\,a = v\texttt{]}, \vec{x}) &= \bigwedge_{1 \le i \le m} x_{i,j}^{s:a} = a_j \wedge x_{i,m+1}^{s:a} = 0 \\ \mathrm{AttsPres}(\texttt{[}s\,|\,a \mathrel{\hat{}{=}} v\texttt{]}, \vec{x}) &= \bigwedge_{1 \le j \le m} x_{i,j}^{s:a} = a_j \\ \mathrm{AttsPres}(\texttt{[}s\,|\,a \star\!= v\texttt{]}, \vec{x}) &= \bigvee_{0 \le j \le N-m-1} \ \bigwedge_{1 \le j' \le m} x_{i,j+j'}^{s:a} = a_j\end{aligned}$$

Finally, we enforce correct use of the null character

$$\mathrm{Nulls}(\vec{x}) = \bigvee_{1 \le j \le N} \ \bigwedge_{j \le j' \le N} x_{i,j'}^{s:a} = 0 \ .$$

[5]Of course, we could obtain individual bounds for each s and a if we wanted to streamline the encoding.
[6] Note, we allow negation in this formula. This is for convenience only as the formulas we negate can easily be transformed into existential Presburger.
[7] Strictly speaking, Presburger variables range over natural numbers. It is straightforward to range over a finite number of values. That is, we can assume, w.l.o.g. that $\Gamma \uplus 0 \subseteq \mathbb{N}$ and the quantification is suitably restricted.

6.4.5 Encoding Non-Emptiness. We are now ready to give the main encoding of the emptiness of a CSS automaton using the quantifier-free theory over integer linear arithmetic. This encoding makes use of a number of variables, which we explain intuitively below. After describing the variables, we give the encoding in two parts: first we explain how a single node selector can be translated into existential Presuburger arithmetic. Once we have this translation, we give the final step of encoding a complete run of an automaton.

Variables Used in the Encoding. Our encoding makes use of the following variables for $0 \leq i \leq n$, representing the node at the ith step of the run. We use the overline notation to indicate variables.

- $\overline{q}_i$, taking any value in Q, indicating the state of the automaton when reading the ith node in the run,
- $\overline{s}_i$, taking any value in $\rfloor(\mathrm{NS})$ indicating the element tag (with namespace) of the ith node read in the run,
- $\overline{e}_i$, taking any value in $\rfloor(E)$ indicating the element tag (with namespace) of the ith node read in the run,
- $\overline{p}_i$, for each pseudo-class $p \in P \setminus \{\texttt{:root}\}$ indicating that the ith node has the pseudo-class p,
- $\overline{n}_i$, taking a natural number indicating that the ith node is the $\overline{n}_i$th child of its parent, and
- $\overline{n}_i^{s:e}$, for all $s \in \rfloor(\mathrm{NS})$ and $e \in \rfloor(E)$, taking a natural number variable indicating that there are $\overline{n}_i^{s:e}$ nodes of type s:e strictly preceding the current node in the sibling order, and
- $\overline{N}_i$, taking a natural number indicating that the current node is the $\overline{N}_i$th to last child of its parent, and
- $\overline{N}_i^{s:e}$, for all $s \in \rfloor(\mathrm{NS})$ and $e \in \rfloor(E)$, taking a natural number variable indicating that there are $\overline{N}_i^{s:e}$ nodes of type s:e strictly following the current node in the sibling order, and
- $x_{i,j}^{s:a}$ as used in the previous section for encoding the character at the jth character position of the attribute value for s:a at position i in the run[8].

Note, we do not need a variable for `:root` since it necessarily holds uniquely at the 0th position of the run.

Encoding Node Selectors. We define the encoding of node selectors below using the variables defined in the previous section. Note, this translation is not correct in isolation: global constraints such as “no ID appears twice in the tree” will be enforced later. The encoding works by translating each part of the selector directly. For example, the constraint e simply checks that $\overline{e}_i = e$. Even in the more complex cases of selectors such as `:nth-child(`α`n + `β`)` we are able to use a rather direct translation of the semantics: $\exists \overline{n}.\overline{x} = \alpha\overline{n} + \beta$. For the case of `:nth-of-type(`$\alpha$`n + `$\beta$`)` we have to consider all possible namespaces s and element names e that the node could take, and use the $\overline{n}_i^{s:e}$ variables to do the required counting.

In our presentation we allow ourselves to negate existentially quantified formulas of the form $\exists \overline{n}.\overline{x} = \alpha\overline{n} + \beta$ where $\overline{x}$ is a variable, and α and β are constants. Although this is not strictly allowed in existential Presburger arithmetic, it is not difficult to encode correctly. For completeness, we provide the encoding of such negated formulas in Appendix C.3.4.

In the following, let $\mathrm{NoAtts}(\Theta)$ be Θ *less* all selectors of the form `[`$s \mid a$`]`, `[`$s \mid a$ op v`]`, `[`a`]`, or `[`a op v`]`, or `:not([s|a])`, `:not([s|a op v])`, `:not([a])`, or `:not([a op v])`.

[8] Recall s is not necessarily in $\rfloor(\mathrm{NS})$ as it may be some fresh value.

Definition 6.6 ($\mathrm{Pres}(\sigma, i)$*).* Given a node selector $\tau\Theta$, we define

$$\mathrm{Pres}(\tau\Theta, i) = \left(\begin{array}{c} \mathrm{Pres}(\tau, i) \wedge \\ \left(\bigwedge\limits_{\theta \in \mathrm{NoAtts}(\Theta)} \mathrm{Pres}(\theta, i) \right) \wedge \\ \mathrm{AttsPres}(\tau\Theta, i) \end{array} \right)$$

where we define $\mathrm{Pres}(\theta, i)$ as follows:

$$\begin{array}{rcl} \mathrm{Pres}(*, i) & = & \top \\ \mathrm{Pres}(\texttt{(}s \,|\, *\texttt{)}, i) & = & (\overline{s}_i = s) \\ \mathrm{Pres}(e, i) & = & (\overline{e}_i = e) \\ \mathrm{Pres}(\texttt{(}s \,|\, e\texttt{)}, i) & = & (\overline{s}_i = s \wedge \overline{e}_i = e) \\ \mathrm{Pres}(\texttt{:not(}\sigma_\neg\texttt{)}, i) & = & \neg\mathrm{Pres}(\sigma_\neg, i) \\ \mathrm{Pres}(\texttt{:root}, i) & = & \begin{cases} \top & i = 0 \\ \bot & \text{otherwise} \end{cases} \\ \forall p \in P \setminus \{\texttt{:root}\} \;.\; \mathrm{Pres}(p, i) & = & \overline{p}_i \end{array}$$

and, finally, for the remaining selectors, we have

$$\begin{array}{rcl} \mathrm{Pres}(\texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)}, 0) & = & \bot \\ \mathrm{Pres}(\texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)}, 0) & = & \bot \\ \mathrm{Pres}(\texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)}, 0) & = & \bot \\ \mathrm{Pres}(\texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)}, 0) & = & \bot \\ \mathrm{Pres}(\texttt{:only-child}, 0) & = & \bot \\ \mathrm{Pres}(\texttt{:only-of-type}, 0) & = & \bot \end{array}$$

and when $i > 0$

$$\begin{array}{rcl} \mathrm{Pres}(\texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)}, i) & = & \exists \overline{n}. \overline{n}_i = \alpha\overline{n} + \beta \\ \mathrm{Pres}(\texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)}, i) & = & \exists \overline{n}. \overline{N}_i = \alpha\overline{n} + \beta \\ \mathrm{Pres}(\texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)}, i) & = & \bigvee\limits_{\substack{s \in \rfloor(\mathrm{NS}) \\ e \in \rfloor(E)}} \left(\begin{array}{c} \overline{s}_i = s \wedge \overline{e}_i = e \wedge \\ \exists \overline{n}. \overline{n}^{s:e}_i + 1 = \alpha\overline{n} + \beta \end{array} \right) \\ \mathrm{Pres}(\texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)}, i) & = & \bigvee\limits_{\substack{s \in \rfloor(\mathrm{NS}) \\ e \in \rfloor(E)}} \left(\begin{array}{c} \overline{s}_i = s \wedge \overline{e}_i = e \wedge \\ \exists \overline{n}. \overline{N}^{s:e}_i + 1 = \alpha\overline{n} + \beta \end{array} \right) \\ \mathrm{Pres}(\texttt{:only-child}, i) & = & \overline{n}_i = 1 \wedge \overline{N}_i = 1 \\ \mathrm{Pres}(\texttt{:only-of-type}, i) & = & \bigvee\limits_{\substack{s \in \rfloor(\mathrm{NS}) \\ e \in \rfloor(E)}} \left(\begin{array}{c} \overline{s}_i = s \wedge \overline{e}_i = e \wedge \\ \overline{n}^{s:e}_i = 0 \wedge \overline{N}^{s:e}_i = 0 \end{array} \right) \end{array}$$

We are now ready to move onto complete the encoding.

Encoding Runs of CSS Automata. Finally, now that we are able to encode attribute and node selectors, we can make use of these to encode accepting runs of a CSS automaton. Since we know, if there is an accepting run, that there is a run of length at most n where n is the number of transitions in Δ, we encode the possibility of an accepting run using the variables discussed above for all $0 \leq i \leq n$. The shape of the translation is given below and elaborated in the sequel.

Definition 6.7. $\theta_{\mathcal{A}}$ Given a CSS automaton $\mathcal{A}$ we define

$$\theta_{\mathcal{A}} = \left(\begin{pmatrix} \overline{q}_0 = q^{\mathrm{in}} \\ \wedge \\ \overline{q}_n = q_f \end{pmatrix} \wedge \bigwedge_{0 \leq i < n} \begin{pmatrix} \mathrm{Tran}(i) \\ \vee \\ \overline{q}_i = q_f \end{pmatrix} \wedge \mathrm{Consistent} \right)$$

where $\mathrm{Tran}(i)$ and $\mathrm{Consistent}$ are defined below.

Intuitively, the first two conjuncts asserts that a final state is reached from an initial state. Next, we use $\mathrm{Tran}(i)$ to encode a single step of the transition relation, or allows the run to finish early. Finally $\mathrm{Consistent}$ asserts consistency constraints.

We define as a disjunction over all possible (single-step) transitions $\mathrm{Tran}(i) = \bigvee_{t \in \Delta} \mathrm{Tran}(i, t)$ where $\mathrm{Tran}(i, t)$ is defined below by cases. There are four cases depending on whether the transition is labelled $\downarrow$, $\rightarrow$, $\rightarrow_+$, or $\circ$. In most cases, we simply assert that the state changes as required by the transition, and that the variables $\overline{n}_i$ and $\overline{n}_i^{s:e}$ are updated consistently with the number of nodes read by the transition. Although the encodings look complex, they are essentially simple bookkeeping.

To ease presentation, we write $\overline{s}_i{:}\overline{e}_i = s{:}e$ as shorthand for $(\overline{s}_i = s \wedge \overline{e}_i = e)$ and $\overline{s}_i{:}\overline{e}_i \neq s{:}e$ as shorthand for $(\overline{s}_i \neq s \vee \overline{e}_i \neq e)$.

(1) When $t = q \xrightarrow[\sigma]{\downarrow} q'$ we define $\mathrm{Tran}(i, t)$ to be

$$\begin{gathered} (\overline{q}_i = q) \wedge \left(\overline{q}_{i+1} = q'\right) \wedge \neg \overline{\texttt{:empty}}_i \wedge \mathrm{Pres}(\sigma, i) \wedge \\ (\overline{n}_{i+1} = 1) \wedge \bigwedge_{\substack{s \in \downharpoonleft(\mathrm{NS}) \\ e \in \downharpoonleft(E)}} \left(\overline{n}_{i+1}^{s:e} = 0\right) . \end{gathered}$$

(2) When $t = q \xrightarrow[\sigma]{\rightarrow} q'$ we define $\mathrm{Tran}(i, t)$ to be false when $i = 0$ (since the root has no siblings) and otherwise

$$\begin{gathered} (\overline{q}_i = q) \wedge \left(\overline{q}_{i+1} = q'\right) \wedge \mathrm{Pres}(\sigma, i) \wedge \\ (\overline{n}_{i+1} = \overline{n}_i + 1) \wedge \left(\overline{N}_{i+1} = \overline{N}_i - 1\right) \wedge \\ \bigwedge_{\substack{s \in \downharpoonleft(\mathrm{NS}) \\ e \in \downharpoonleft(E)}} \begin{pmatrix} \left((\overline{s}_i{:}\overline{e}_i = s{:}e) \Rightarrow \left(\overline{n}_{i+1}^{s:e} = \overline{n}_i^{s:e} + 1\right)\right) \wedge \\ \left((\overline{s}_i{:}\overline{e}_i \neq s{:}e) \Rightarrow \left(\overline{n}_{i+1}^{s:e} = \overline{n}_i^{s:e}\right)\right) \wedge \\ \left((\overline{s}_{i+1}{:}\overline{e}_{i+1} = s{:}e) \Rightarrow \left(\overline{N}_{i+1}^{s:e} = \overline{N}_i^{s:e} - 1\right)\right) \wedge \\ \left((\overline{s}_{i+1}{:}\overline{e}_{i+1} \neq s{:}e) \Rightarrow \left(\overline{N}_{i+1}^{s:e} = \overline{N}_i^{s:e}\right)\right) \end{pmatrix} . \end{gathered}$$

(3) When $t = q \xrightarrow[*]{\rightarrow_+} q$ we define $\mathrm{Tran}(i, t)$ to be false when $i = 0$ and otherwise

$$\begin{gathered} (\overline{q}_i = q) \wedge \left(\overline{q}_{i+1} = q\right) \wedge \\ \exists \overline{\delta}. \left(\left(\overline{n}_{i+1} = \overline{n}_i + \overline{\delta}\right) \wedge \left(\overline{N}_{i+1} = \overline{N}_i - \overline{\delta}\right)\right) \wedge \\ \bigwedge_{\substack{s \in \downharpoonleft(\mathrm{NS}) \\ e \in \downharpoonleft(E)}} \exists \overline{\delta}_{s:e} . \begin{pmatrix} \begin{pmatrix} (\overline{s}_i{:}\overline{e}_i = s{:}e) \Rightarrow \\ \left(\overline{n}_{i+1}^{s:e} = \overline{n}_i^{s:e} + \overline{\delta}_{s:e} + 1\right) \end{pmatrix} \wedge \\ \begin{pmatrix} (\overline{s}_i{:}\overline{e}_i \neq s{:}e) \Rightarrow \\ \left(\overline{n}_{i+1}^{s:e} = \overline{n}_i^{s:e} + \overline{\delta}_{s:e}\right) \end{pmatrix} \wedge \\ \begin{pmatrix} (\overline{s}_{i+1}{:}\overline{e}_{i+1} = s{:}e) \Rightarrow \\ \left(\overline{N}_{i+1}^{s:e} = \overline{N}_i^{s:e} - \overline{\delta}_{s:e} - 1\right) \end{pmatrix} \wedge \\ \begin{pmatrix} (\overline{s}_{i+1}{:}\overline{e}_{i+1} \neq s{:}e) \Rightarrow \\ \left(\overline{N}_{i+1}^{s:e} = \overline{N}_i^{s:e} - \overline{\delta}_{s:e}\right) \end{pmatrix} \end{pmatrix} . \end{gathered}$$

(4) When $t = q \xrightarrow[\sigma]{\circ} q'$ we define $\mathrm{Tran}(i, t)$ to be

$$(\overline{q}_i = q) \wedge \left(\overline{q}_{i+1} = q'\right) \wedge \mathrm{Pres}(\sigma, i)\ .$$

To ensure that the run is over a consistent tree, we assert the consistency constraint

$$\mathrm{Consistent} = \mathrm{Consistent}_n \wedge \mathrm{Consistent}_i \wedge \mathrm{Consistent}_p$$

where each conjunct is defined below.

- The clause $\mathrm{Consistent}_n$ asserts that the values of $\overline{n}_i$, $\overline{N}_i$, $\overline{n}_i^{s:e}$, and $\overline{N}_i^{s:e}$ are consistent. That is
$$\bigwedge_{1 \leq i \leq n} \left(\overline{n}_i = 1 + \sum_{s:e \in E} \overline{n}_i^{s:e}\right) \wedge \left(\overline{N}_i = 1 + \sum_{s:e \in E} \overline{N}_i^{s:e}\right)\ .$$
- The clause $\mathrm{Consistent}_i$ asserts that ID values are unique. It is the conjunction of the following clauses. For each s for which we have created variables of the form $x_{i,j}^{s:\mathtt{id}}$ we assert
$$\bigwedge_{1 \leq i \neq i' \leq n} \bigvee_{1 \leq j \leq N} x_{i,j}^{s:\mathtt{id}} \neq x_{i',j}^{s:\mathtt{id}}\ .$$
- Finally, $\mathrm{Consistent}_p$ asserts the remaining consistency constraints on the pseudo-classes. We define $\mathrm{Consistent}_p =$
$$\bigwedge_{0 \leq i \leq n} \left(\begin{array}{c} \neg\left(\overline{\mathtt{:link}_i} \wedge \overline{\mathtt{:visited}_i}\right) \wedge \\ \bigwedge_{0 \leq j \neq i \leq n} \left(\neg\left(\overline{\mathtt{:target}_i} \wedge \overline{\mathtt{:target}_j}\right) \right) \wedge \\ \neg\left(\overline{\mathtt{:enabled}_i} \wedge \overline{\mathtt{:disabled}_i}\right) \end{array} \right)\ .$$
These conditions assert the mutual exclusivity of `:link` and `:visited`, that at most one node in the document can be the target node, that nodes are not both enabled and disabled.

6.4.6 Correctness of the Encoding. We have now completed the definition of the reduction from the emptiness problem of CSS automata to the satisfiability of existential Presburger arithmetic. What remains is to show that this reduction is faithful: that is, the CSS automaton has an empty language if and only if the formula is satisfiable. The proof is quite routine, and presented in Lemma C.8 and Lemma C.9 in Appendix C.3.5.

LEMMA 6.8 (CORRECTNESS OF $\theta_{\mathcal{A}}$). *For a CSS automaton $\mathcal{A}$, we have*

$$\mathcal{L}(\mathcal{A}) \neq \emptyset \Leftrightarrow \theta_{\mathcal{A}} \text{ is satisfiable.}$$

We are thus able to decide the emptiness problem, and therefore the emptiness of intersection problem, for CSS automata by reducing the problem to satisfiability of existential Presburger arithmetic and using a fast solver such as Z3 (de Moura and Bjørner 2008) to resolve the satisfiability.

7 REFACTORING TO MAX-SAT

In this section, we provide a reduction from the refactoring problem to partial weighted MaxSAT. The input will be a valid covering $C = \{\overline{B}_i\}_{i=1}^m$ of a CSS graph $\mathcal{G}$. We aim to find a rule $\overline{B} = (X, \overline{Y})$ and a position j that minimises the weight of $C[\overline{B} \to j]_{\downarrow}$.

There is a fairly straightforward encoding of the refactoring problem into Max-SAT using for each node $w \in S \cup P$ a boolean variable $\overline{w}$ which is true iff the node is included in the new rule. Unfortunately, early experiments showed that such an encoding does not perform well in practice, causing prohibitively long run times even on small examples. The failure of the naive encoding may be due to the search space that includes a large number of possible pairs $(X, \overline{Y})$ that turn out to be invalid rules (e.g. include edges not in the CSS-graph $\mathcal{G}$). Hence, we will use a different Max-SAT encoding that, by means of syntax, further restricts the search space of valid refactorings.

The crux of our new encoding is to explicitly say in the Max-SAT formula φ that the rule $\overline{B}$ in a refactoring opportunity $(\overline{B}, j)$ is a "valid" sub-biclique of one of the *maximal* bicliques $B = (X, Y)$ — maximal with respect to subset-of relations of X and Y components of bicliques — in the CSS-graph $\mathcal{G}$. By insisting $\overline{B}$ is contained within a maximal biclique of the CSS-graph, we automatically ensure that $\overline{B}$ does not contain edges that are not in $\mathcal{G}$.

The formula φ will try to guess a maximal biclique B and which nodes to omit from B. Since the number of maximal bicliques in a bipartite graph is exponential (in the number of nodes) in the worst case, one concern with this idea is that the constraint φ might become prohibitively large. As we shall see, this turns out not to be the case in practice. Intuitively, based on our experience, the number of rules in a real-world CSS file is in the order of magnitude of 1000 *at most*. Secondly, the number of maximal bicliques in a CSS-graph that corresponds to a real-world CSS file also is typically of a similar size to the number of rules, and, furthermore, can be enumerated using the algorithm from (Kayaaslan 2010) (which runs in time polynomial in the size of the input and the size of the output). To be more precise, the benchmarks in our experiments had between 31 and 2907 rules, and the mean number of rules was 730. The mean ratio of the number of maximal bicliques to the number of rules was 1.25, and the maximum was 2.05. As we shall see in Section 8, Z3 may solve the constraints via this encoding quite efficiently.

In the rest of the section, we will describe our encoding in detail. For convenience, our encoding also allows bounded integer variables. There are standard ways to encode these in binary as booleans (e.g. see (Petke 2015)) by bit-blasting (using a logarithmic number of boolean variables).

7.1 Orderable Bicliques

Our description above of the crux of our encoding (by restricting to containment in a maximal biclique) is a simplification. This is because *not all* sub-bicliques of a maximal biclique correspond to a valid rule $\overline{B}$ in a refactoring opportunity $(\overline{B}, j)$ with respect to the covering C. [A biclique (X', Y') is a *sub-biclique* of a biclique (X, Y) if $X' \subseteq X$ and $Y' \subseteq Y$.] To ensure that our constraint φ chooses only valid rules, it needs to ensure that the sub-biclique that is chosen is "orderable". More precisely, a biclique $B = (X, Y)$ is *orderable at position* j if it can be turned into a rule $\overline{B} = (X, \overline{Y})$ (i.e. turning the set Y of declarations into a sequence $\overline{Y}$ by assigning an order) that can be inserted at position j in C without violating the validity of the resulting covering with respect to the order $\prec$ (from the CSS-graph $\mathcal{G}$). If there are m rules in C, there are $m + 1$ positions (call these positions $0, \ldots, m$) where $\overline{Y}$ may be inserted into C. We show below that the position j is crucial to whether $\overline{B}$ is orderable.

Unorderable bicliques rarely arise in practice (in our benchmarks, the mean percentage of maximal bicliques that were unorderable at some position was 0.44%), but they have to be accounted for if our analysis is to find the optimal refactoring. A biclique $B = (X, Y)$ is unorderable when they have the same property name (or two related property names, e.g., shorthands) occuring multiple times with different values in Y. One reason having a CSS rule with the same property name occuring multiple times with different values is to provide "fallback options" especially because old browsers may not support certain values in some property names, e.g., the rule

```
    .c { color:#ccc; color:rgba(0, 0, 0, 0.5); }
```

says that if `rgba()` is not supported (e.g. in IE8 or older browsers), then use `#ccc`. Using this idea, we can construct the simple example of an unorderable biclique in the CSS file in Figure 7.

```
      .a { color:blue; color:green }
      .b { color:green; color:blue }
```

Fig. 7. A CSS file with an unorderable sub-biclique.

The ordering constraints we can derive from this file include

$$(\texttt{.a}, \texttt{color:blue}) \prec (\texttt{.a}, \texttt{color:green})$$

and

$$(\texttt{.b}, \texttt{color:green}) \prec (\texttt{.b}, \texttt{color:blue})\,.$$

The biclique B

$$(\{\texttt{.a}, \texttt{.b}\}, \{\texttt{color:blue}, \texttt{color:green}\})$$

is easily seen to be orderable at position 0 and 1. This is because the final rule in Figure 7 will ensure the ordering $(\texttt{.b}, \texttt{color:green}) \prec (\texttt{.b}, \texttt{color:blue})$ is satisfied, and since B will appear before this final rule, only $(\texttt{.a}, \texttt{color:blue}) \prec (\texttt{.a}, \texttt{color:green})$ needs to be maintained by B (in fact, at position 0, neither of the orderings need to be respected by B). However, at position 2, which is at the end of the file in Figure 7, both orderings will have to be respected by B. Unfortunately, one of these orderings will be violated regardless of how one may assign an ordering to `blue` and `green`. This contrived example was made only for illustration, however, our technique should still be able to handle even contrived examples.

We mention that both checking orderability and ordering a given biclique can be done efficiently.

PROPOSITION 7.1. *Given a biclique B, a covering C (with m rules) of a CSS-graph $\mathcal{G}$, and a number $j \in \{0, \ldots, m\}$, checking whether B is orderable at position j in C can be done in polynomial time. Moreover, if B is orderable an ordering can be calculated in polynomial time.*

The proof of the proposition is easy and is relegated into Appendix A.1.

Maximal Orderable Bicliques. Our Max-SAT encoding φ needs to ensure that we only pick a pair (B, j) such that B is an orderable biclique at position j in the given covering C, i.e., B corresponds to a rule that can be inserted at position j in C. Although the check of orderability can be *declaratively* expressed as a constraint in φ, we found that this results in Max-SAT formulas that are rather difficult to solve by existing Max-SAT solvers. For this reason, we propose to express the check of orderability in a different way. Intuitively, for each $j \in \{0, \ldots, m\}$, we enumerate all orderable bicliques $B = (X, Y)$ that are also maximal, i.e., it is *not* a (strict) sub-biclique of a different orderable biclique. Since "orderability is inherited by sub-bicliques" (as the following lemma, whose proof is immediate from the definition, states), the constraint φ needs to simply choose a sub-biclique of a maximal orderable biclique that appears in our enumeration.

LEMMA 7.2. *Every sub-biclique $B' = (X, Y)$ of an orderable biclique $B = (X, Y)$ is orderable.*

The above enumeration of maximal orderable bicliques can be described as a pair $(\{M_i\}_{i=1}^{\mu}, \mathcal{F})$ where

- $\{M_i\}_{i=1}^{\mu}$ is an enumeration of all bicliques that are orderable and maximal at some position j, and
- $\mathcal{F}$ *forbids* certain bicliques at each position. I.e. it is a function from $[1, m]$ to the set of bicliques in $\{M_i\}_{i=1}^{\mu}$ that are unorderable at position j.

Observe that the set of orderable bicliques at position j in C is a subset of the set of orderable bicliques at position $j + 1$ in C. This may be formally expressed as: $\mathcal{F}(j) \subseteq \mathcal{F}(j + 1)$ for all $j \in [1, m)$.

In the majority (54%) of examples that we have from real-world CSS, the function $\mathcal{F}$ maps all values of $[1, m]$ to $\emptyset$, i.e., all maximal bicliques are orderable at all positions. The mean percentage of maximal bicliques that were unorderable at some position was 0.44%, with a maximum of 5.84%.

In our description of the Max-SAT encoding below, we assume that the pair $(\{M_i\}_{i=1}^{\mu}, \mathcal{F})$ has been computed for the input C.

7.2 The Max-SAT Encoding

We present the full reduction of the refactoring problem to Max-SAT. In particular, the constraints we produce are

$$(\Pi_H, \Pi_S)$$

where Π_H and Π_S are, respectively, hard and soft constraints. First, we describe the variables used in our encoding. Note, our encoding will rely on the assumption that covering C has already been trimmed.

7.2.1 Representing the Refactoring. We need to represent a refactoring $(\overline{B}, j)$. We use a bounded integer variable $\overline{j}$ (with range $0 \leq \overline{j} \leq m$) to encode j.

For $\overline{B}$ we select a biclique in $\{M_i\}_{i=1}^{\mu}$ and remove some nodes to produce as small a refactoring as possible. We use a bounded integer variable $\overline{i}_M$ (with range $[1, \mu]$) to select M_i. Next, we need to choose a sub-biclique of M_i, which can be achieved by choosing nodes M_i to be removed. To minimise the number of variables used, we number the nodes contained in each biclique in some (arbitrary) way, i.e., for each $i \in [1, \mu]$ and biclique $M_i = (X, Y)$, we define a bijection $\rho_i : X \cup Y \rightarrow [1, |X \cup Y|]$. Let χ be the maximum number of nodes in a biclique M_i in the enumeration $\{M_i\}_{i=1}^{\mu}$, i.e., the maximal integer k such that $\rho_i(w) = k$ for some $w \in S \cup P$ and $1 \leq i \leq \mu$.

We introduce boolean variables $\overline{x}_1, \ldots, \overline{x}_{\chi}$. Once the maximal orderable biclique M_i is picked, for a node w with $\rho_i(w) = k$, the variable $\overline{x}_k$ is used to indicate that w is to be excluded from selected M_i (i.e. $\overline{x}_k$ is true iff w is to be excluded from M_i). More precisely, for an edge $e = (s, p)$ in M_i, we define a predicate $\mathrm{HasEdge}(e)$ to be

$$\mathrm{HasEdge}((s, p)) = \bigvee_{1 \leq i \leq \mu} \overline{i}_M = i \wedge \neg \overline{x}_{\rho_i(s)} \wedge \neg \overline{x}_{\rho_i(p)} \ .$$

Note, when $M_i = (X, Y)$ and $w \notin X \cup Y$ we let $\overline{x}_{\rho_i(w)}$ denote the formula "true".

7.2.2 Hard Constraints. We define

$$\Pi_H = \{\varphi_{\mathrm{vld}}, \varphi_{\mathrm{ord}}\}$$

where φ_{vld} and φ_{ord} are described below.

We need to ensure the refactoring is valid, i.e., it has not been forbidden at the chosen position and inserting it into the covering does not violate the edge order $\lessdot$. For the former, we define $\mathcal{F}^{1\mathrm{st}}$, which is used to discover which of the M_i first become unorderable at position j. That is $M_i \in \mathcal{F}^{1\mathrm{st}}(j)$ if j is the smallest integer such that $M_i \in \mathcal{F}(j)$.

$$\varphi_{\mathrm{vld}} = \bigwedge_{1 \leq j \leq m} \left(\left(\overline{j} >= j\right) \Rightarrow \bigwedge_{M_i \in \mathcal{F}^{1\mathrm{st}}(j)} \left(\overline{i}_M \neq i\right) \right) .$$

We also need to ensure the edge ordering is respected by the refactoring, for which we define φ_{ord}. If $e_1 \lessdot e_2$, and $e_1 = (s_1, p_1)$ is in the rule $\overline{B}$ in the guessed refactoring opportunity $(\overline{B}, j)$, then we need to assert $e_1 \lessdot e_2$ is respected. It is respected either if $e_2 = (s_2, p_2)$ appears after the position where j is to be inserted in C, or e_2 is also contained in the new rule $\overline{B}$ (in which case the ordering can still be respected since $\overline{B}$ is orderable). That is,

$$\varphi_{\mathrm{ord}} = \left(\bigwedge_{(s_1,p_1) \prec (s_2,p_2)} \mathrm{HasEdge}((s_1,p_1)) \Rightarrow \left(\bar{j} < \mathrm{index}((s_2,p_2)) \vee \mathrm{HasEdge}((s_2,p_2))\right) \right) .$$

This is because only the last occurrence of an edge in a covering matters (recall the definition of index of an edge in Section 4). Also note that our use of bicliques ensures that we do not introduce pairs (s,p) that are not edges in E.

7.2.3 Soft Constraints. The soft constraints will calculate the weight of $C[\overline{B} \to j]_{\downarrow}$. Since we want to minimise this weight, the optimal solution to the Max-SAT problem will give an optimal refactoring. Our soft constraints are

$$\Pi_S = \Pi_S^{\overline{B}} \cup \Pi_S^{\mathrm{Sels}} \cup \Pi_S^{\mathrm{Props}}$$

where $\Pi_S^{\overline{B}}$ counts the weight of the $\overline{B}$, and Π_S^{Sels} and Π_S^{Props} counts the weight of the remainder (not including $\overline{B}$) of the stylesheet by counting the remaining selectors and properties respectively after trimming. These are defined below.

To count the weight of $\overline{B}$, we count the weight of the non-excluded nodes of $M_i = (X_i, Y_i)$. We have

$$\Pi_S^{\overline{B}} = \left\{ (\varphi_w^i, \mathtt{wt}(w)) \;\middle|\; 1 \le i \le \mu \wedge w \in X_i \cup Y_i \right\}$$

where

$$\varphi_w^i = \left(\left(\bar{i}_M = i \right) \Rightarrow \overline{x}_{\rho_i(w)} \right) .$$

Note that φ_w^i is true iff, whenever M_i is picked, the node w is omitted from M_i in the guessed $\overline{B}$. Furthermore, the cost of *not* omitting w from M_i is $\mathtt{wt}(w)$.

Next, we count the remaining weight of $C[\overline{B} \to j]$ (i.e. excluding the weight of $\overline{B}$). Assume that C has already been trimmed, i.e., $C_{\downarrow} = C$. The intuition is that a node v can be removed from $\overline{B}_i$ in C if all edges e incident to v appear in a later rule in C, i.e., $\mathrm{index}(e) > i$. In particular, let $\overline{B}_i = (X_i, \overline{Y}_i)$ in $\{\overline{B}_i\}_{i=1}^m$. To count the weight of the untrimmed selectors we use the clauses

$$\Pi_S^{\mathrm{Sels}} = \left\{ (\psi_s^i, \mathtt{wt}(s)) \;\middle|\; 1 \le i \le m \wedge s \in X_i \right\}$$

where

$$\psi_s^i = \left(i \le \bar{j} \wedge \bigwedge_{\substack{\mathrm{index}((s,p))=i \\ p \in \overline{Y}}} \mathrm{HasEdge}((s,p)) \right) .$$

The idea is that ψ_s^i will be satisfied whenever s can be removed from $\overline{B}_i$. We assume C has already been trimmed, so s can only become removable because of the refactoring. This explains the first conjunct which asserts that nodes can only be removed from rules appearing before $\bar{j}$. Next, s can be removed after refactoring if none of its incident edges (s,p) are the final occurrence of (s,p) in $C[\overline{B} \to j]$. The crucial edges in this check are those such that $\mathrm{index}((s,p)) = i$, which means $\overline{B}_i$ contains the final occurrence of (s,p) in C. For these edges, if $\mathrm{HasEdge}((s,p))$ holds, then the new rule contains (s,p) and $\overline{B}_i$ will no longer contain the final occurrence of (s,p) after refactoring.

Similarly, to count the weight of the properties that cannot be removed we have

$$\Pi_S^{\mathrm{Props}} = \left\{ (\psi_p^i, \mathtt{wt}(p)) \;\middle|\; 1 \le i \le m \wedge p \in \overline{Y}_i \right\}$$

where

$$\psi_p^i = \left(i \leq \bar{j} \wedge \bigwedge_{\substack{\text{index}((s,p))=i \\ s \in X}} \text{HasEdge}((s,p)) \right).$$

7.3 Generated Refactoring

The refactoring opportunity $(\overline{B}, j)$ is built from an optimal satisfying assignment to (Π_H, Π_S). First, j is the value assigned to $\bar{j}$. Then $\overline{B} = (X, \overline{Y})$ where, letting i_M be the value of $\bar{i}_M$, and letting $M_{i_M} = (X', Y')$,

- $s \in X$ iff $s \in X'$ and $\bar{x}_{\rho_{i_M}(s)}$ is assigned the value false, and
- $\overline{Y} = \{p_i\}_{i=1}^m$, where $\{p_i\}_{i=1}^m$ is obtained by assigning an ordering to Y' such that $(\overline{B}, j)$ is a valid refactoring.

That the ordering of Y' above exists is guaranteed by the fact that M_{i_M} is orderable at j, and Lemma 7.2. We can compute the ordering in polynomial time via Proposition 7.1. We argue the following proposition in Appendix A.3.

PROPOSITION 7.3. *The refactoring* $(\overline{B}, j)$ *generated from the maximal solution to* (Π_H, Π_S) *is the optimal refactoring of* C.

8 EXPERIMENTAL RESULTS

We implemented a tool SATCSS (in Python 2.7) for CSS refactoring which can be found in our supplementary material. The tool is also available as source code and in a working disk image via the following URLs.

https://github.com/matthewhague/sat-css-tool
http://www.cs.rhul.ac.uk/home/hague/sat-css-tool.img.txz

It constructs the edge order following Section 5 and discovers refactoring following Section 7. We use Z3 (de Moura and Bjørner 2008) as the back end SMT and Max-SAT solver. As an additional contribution, our tool can also generate instances in the extended DIMACS format (Argelich et al. 2016) allowing us to use any Max-SAT solvers. This output may also provide a source of industrially inspired examples for Max-SAT competition benchmarks.

Our benchmarks comprise 71 CSS files coming from three different sources (see Appendix D.2 for a complete listing):

- We collected CSS files from the top 20 websites on a global ranking list (Alexa Internet 2017).
- CSS files were taken from a further 12 websites selected from the top 20-65 websites from the listing above.
- Finally, CSS files were taken from 10 smaller websites such as DBLP. These were examples used during the development of the tool.

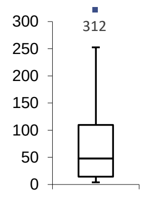


Fig. 8. Box plot of the file sizes in kilobytes

This selection gives us a wide-range of different types of websites, including large scale industrial websites and those developed by smaller teams. Note, several websites contained more than one CSS file and in this case we took a selection of CSS files from the site. Hence, we collected more examples than the number of websites used. Figure 8 gives the file-size distribution of the CSS files we collected.

In the following, we describe the optimisations implemented during the development of SATCSS. We then describe the particulars of the experimental setup and provide the results in Figure 9.

8.1 Optimisations

We give an overview of the optimisations we used when implementing SATCSS.

- We introduce variables $\overline{x}_k$ only for nodes appearing in the edge order, rather than for all nodes in a biclique. This is enough to be able to define sub-bicliques of any M_i that can appear at any position in the covering, but means that the smallest refactoring cannot always be constructed. However, this reduces the search space and leads to a significant improvement to run times.
- For a refactoring to reduce the size of the file, it must remove at least two nodes from the covering. Hence, for each M_i let j^i_l be the index of the *second* rule in C containing some edge in M_i (not necessarily the same edge), and j^i_h be the index of the *last* rule containing some edge in M_i Without loss of generality we assert

$$\bar{i}_M = i \Rightarrow \left(\bar{j} \geq j^i_l \wedge \bar{j} \leq j^i_h\right) \ .$$

- We performed the following optimisation when calculating $(\{M_i\}^{\mu}_{i=1}, \mathcal{F})$. The majority (54%) of benchmarks all maximal bicliques are orderable at all positions, and the mean percentage of maximal bicliques that were unorderable at some position was 0.44%, with a maximum of 5.84%. However, there are one or two of the largest examples of our experiments where this enumeration took a minute or so. Since the number of maximal bicliques that are unorderable at some position is so small, we decided in our implementation to simply remove all such bicliques from the analysis. In this case $\{M_i\}^{\mu}_{i=1}$ is an enumeration of all maximal bicliques that are orderable at all positions j, and $\mathcal{F}$ maps all values of $[1, m]$ to $\emptyset$. This means that we may not be able find the optimal refactoring as not all bicliques are available, but since the amount of time required to find a refactoring is reduced, we are able to find more refactorings to apply.
- Finally, we allowed a multi-threaded partitioned search. This works as follows. The search space is divided across n threads, and each thread partitions its search space into m partitions. During iteration j, thread k allows only those M_i where $i = k * m + (j \mod m)$. If the fastest thread finds a refactoring in t seconds, we wait up to $0.1t$ seconds for further instances. We take the best refactoring of those that have completed. A thread reports "no refactoring found" only if none of its partitions contain a refactoring.

  For the experiments we implemented a simple heuristic to determine the number of threads and partitions to use. We describe this heuristic here, but first note that better results could likely be obtained with systematic parameter tuning rather than our ad-hoc settings. The heuristic used was to count the number of nodes in the CSS file; that is, the total number of selectors and property declarations (this total includes repetitions of the same node – we do not identify repetitions of the same node). The tool creates enough partitions to give up to 750 nodes per partition. If only two partitions are needed, only one thread is used. Otherwise SATCSS first creates new threads – up to the total number of CPUs on the machine. Once this limit is reached, each thread is partitioned further until the following holds: $\text{number of threads} * \text{partitions per threads} * 750 \leq \text{number of nodes}$.

For the edge ordering:

- We do not support attribute selectors in full, but perform simple satisfiability checks on the most commonly occurring types of constraints. These cover all constraints in our benchmarks. However, we do support shorthand properties.

- Instead of doing a full Existential Presburger encoding, we do a backwards emptiness check of the CSS automata. This backwards search collects smaller Existential Presburger constraints describing the relationship between siblings in the tree, and checks they are satisfiable before moving to a parent node. Global constraints such as ID constraints are also collected and checked at the root of the tree. This algorithm is described in Appendix D.1.

8.2 Results

The experiments were run on a Dell Latitude e6320 laptop with 4Gb of RAM and four 2.7GHz Intel i7-2620M cores. The Python interpreter used was PyPy 5.6 (Rigo and Contributors 2007) and the backend version of Z3 was 4.5. Each experiment was run up to a timeout of 30 minutes. In the case where CSS files used media queries (which our techniques do not yet support), we used stripmq (Hickenlooper 2014) with default arguments to remove the media queries from CSS files. Also, we removed whitespaces and comments from all the CSS files before they were processed by the minifiers and our tool.

We used a timeout of 30 minutes because minification only needs to be applied once before deployment of the website. We note that the tool finds many refactorings during this period and a minified stylesheet can be produced after each refactoring. This means the user will be able to trade time against the amount of size reduction made. Moreover, refactorings tend to show a "long-tail" where the first refactorings found provide larger savings and the later refactorings show dimishing returns.

Table 1 and Figure 9 summarise the results. We compared our tool with six popular minifiers in current usage (Slant 2017). There are two groups of results: the first set show the results when either our tool or one of the minifiers is used alone on each benchmark; the second show the results when the CSS files are run first through a minifier and then through our tool. This second batch of results shows a significant improvement over running single tools in isolation. Thus, our tool complements and improves existing minification algorithms and our techniques may be incorporated into a suite of minification techniques.

Table 1 shows the savings, after whitespaces and comments are removed, obtained by SatCSS and the six minifiers when they are used alone and together. The table presents the savings in seven percentile ranks that include the minimal (0th), the median (50th), and the maximal values (100th). The upper half of the table shows the savings obtained in bytes, while the lower half shows the savings as percentages of the original file sizes. The first seven columns in the table show the savings when either our tool or one of the minifiers is used alone; the rest of the columns show the results when the CSS files are processed first by a minifier and then by our tool. Figure 9 shows the same data in visual form. It can be seen that SatCSS tends to achieve greater savings than each of the six minifiers on our benchmarks. Furthermore, even greater savings can be obtained when SatCSS is used in conjunction with any of the six minification tools. More precisely, when run individually, SatCSS achieves savings with a third quartile of 6.77% and a median value of 3.91%, while the six minifiers achieve savings with third quantiles and medians up to 5.87% and 3.24%, respectively. When we run SatCSS after running any one of these minifiers, the third quartile of the savings can be increased to 8.30% and the median to 5.11%. The additional gains obtained by SatCSS on top of the six minifiers (as a percentage of the original file size) have a third quartile of 5.19% and a median value of 3.22%. Moreover, the ratios of the percentage of savings made by SatCSS to the percentage of savings made by the six minifiers have third quartiles of at least 144% and medians of at least 63%. These figures clearly indicate a substantial proportion of extra space savings made by SatCSS.

tool / percentile	satcss	csso	cssnano	cleancss	minify	yui	cssmin	csso satcss	cssnano satcss	cleancss satcss	minify satcss	yui satcss	cssmin satcss
0th	14	-70	0	11	0	-182	0	33	27	33	22	22	22
20th	342	268	71	242	6	2	4	447	627	630	365	368	371
40th	1082	921	184	791	28	18	24	1696	1630	1759	1235	1174	1232
50th	2158	1525	380	1280	68	46	52	4055	2964	4392	2587	2505	2613
60th	2829	2909	658	2223	154	128	117	5138	3958	4976	3549	3236	3497
80th	4826	4795	3224	5417	267	238	210	7824	7701	8886	5196	4975	5260
100th	86293	86112	24047	69629	3727	3551	3224	102663	85324	90274	85728	85774	85302

tool / percentile	satcss	csso	cssnano	cleancss	minify	yui	cssmin	csso satcss	cssnano satcss	cleancss satcss	minify satcss	yui satcss	cssmin satcss
0th	0.08	-0.55	0.00	0.12	0.00	-0.42	0.00	0.14	0.31	0.35	0.08	0.09	0.09
20th	1.84	1.45	0.20	1.26	0.01	0.00	0.01	3.25	2.88	3.73	2.03	2.12	2.04
40th	3.28	2.58	0.63	2.59	0.09	0.08	0.08	4.89	4.56	5.27	3.68	3.63	3.68
50th	3.91	3.24	1.68	3.10	0.15	0.13	0.17	5.77	5.59	6.78	4.23	4.18	4.23
60th	4.73	4.31	2.26	3.67	0.22	0.22	0.21	6.71	6.82	7.23	5.01	5.11	5.13
80th	7.87	5.58	4.93	6.62	0.55	0.50	0.48	10.14	11.01	10.96	8.24	7.93	7.90
100th	27.68	27.62	23.26	25.68	6.72	6.40	3.56	32.93	30.90	29.72	27.50	27.52	27.36

Table 1. Percentile ranks of the savings in bytes (above) and in percentages (below)

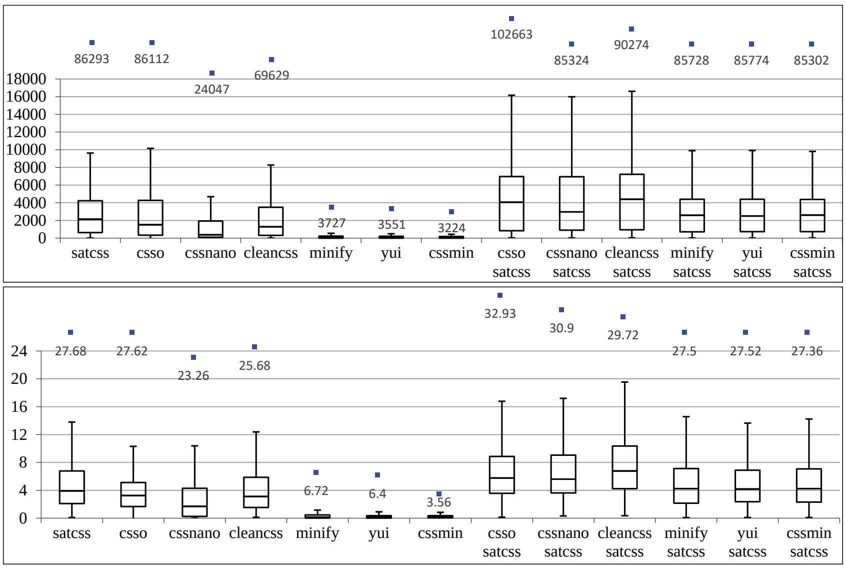


Fig. 9. Box plots of the savings in bytes (above) and in percentages (below)

9 RELATED WORK

CSS minification started to receive attention in the web programming community around the year of 2000. To the best of our knowledge, the first major tools that could perform CSS minification were Yahoo! YUI Compressor (Sha and Contributors 2014) and Microsoft Ajax Minifier, both of which were developed around 2004–2006. This is followed by the development of many other CSS minifiers including (in no particular order) `cssmin` (Bleuzen 2017), `clean-css` (Pawlowicz 2017), `csso` (Dvornov and Contributors 2017), `cssnano` (Briggs and Contributors 2015), and `minify` (Clay and Contributors 2017). Such minifiers mostly apply syntactic transformations including removing whitespace characters, comments, and replacing strings by their abbreviations (e.g. `#f60` by `#ff6600`). More and more advanced optimisations are also being developed. For example, `cssnano` provides a limited support of our rule refactoring, wherein only *adjacent* rules may be merged or combined. The lack of techniques for handling the order dependencies of CSS rules (Souders 2007) was most likely one main reason why a more general rule refactoring (e.g. that can merge or combine rules that are far away in the file) is not supported by CSS minifiers.

Although the importance of CSS minification is understood in industry, the problem received little attention in academia until very recently. Below we will mention the small number of existing work on formalisation of CSS selectors and CSS minification, and other relevant work.

The lack of theories for reasoning about CSS selectors was first mentioned in the paper (Genevès et al. 2012), wherein the authors developed a tree logic for algorithmically reasoning about CSS selectors, i.e., by developing algorithms for deciding satisfiability of formulas in the logic. This formalisation does *not* capture the whole class of CSS3 selectors; as remarked in their follow-up paper (Bosch et al. 2015), the logic captures 70% of the selectors in their benchmarks from real-world CSS files. In particular, they do not fully support attribute selectors (e.g. `[l*="bob"]`). Our paper provides a *full* formalisation of CSS3 selectors. In addition, their tree logic can express properties that are *not* expressible in CSS. Upon a closer inspection, their logic is at least as expressive as μ-calculus, which was a well-known logic in database theory for formalising query languages over XML documents, e.g., see (Libkin 2006; Neven 2002) for two wonderful surveys. As such, the complexity of satisfiability for their tree logic is EXPTIME-hard. Our formalisation captures *no more* than the expressive power of CSS3 selectors, which helps us obtain the much lower complexity NP and enables the use of highly-optimised SMT-solvers. There is a plethora of other work on logics and automata on unranked trees (with and without data), e.g., see (Benedikt et al. 2005, 2003; David et al. 2012; Figueira 2010; Geerts and Fan 2005; Genevès and Layaïda 2006; Genevès et al. 2015; Gottlob and Koch 2002; Hidders 2003; Libkin and Sirangelo 2010; Marx 2005; Marx and de Rijke 2005; Neven and Schwentick 2006; Seidl et al. 2004; ten Cate et al. 2010; ten Cate and Marx 2007, 2009) and the surveys (Bojańczyk 2010; Libkin 2006; Neven 2002). However, none of these formalisms can capture certain aspects of CSS selectors (e.g. string constraints on attribute values), even though they are much more powerful than CSS selectors in other aspects (e.g. navigational).

There is a handful of research results on CSS minification and refactoring that appeared in recent years. Loosely speaking, these optimisations can be categorised into two: (a) *document-independent*, and (b) *document-dependent*. Document-independent optimisations are program transformations that are performed completely independently of the web documents (XML, HTML, etc.). On the other hand, document-dependent optimisations are performed with respect to a (possibly infinite) set of documents. Existing CSS minifiers *only* perform document-independent optimisations since they are meant to preserve the semantics of the CSS file *regardless* of the HTML document to which the CSS file is applied. Our work in this paper falls within this category too. Such optimisations are often the most sensible option *in practice* including (1) the case of *generic* stylesheets as part of web templates (e.g. WordPress), and (2) the case when the stylesheets are used as part of an HTML5 application with JavaScript employed. Case (2) requires some explanation. A typical HTML5 application comes with a finite set of HTML documents, JavaScript code, and CSS files. The presence of JavaScript means that potentially *infinitely* many possible DOM-trees could be generated and displayed by the browser. Therefore, a CSS minification/refactoring should *not* affect the rendering of *any* such tree by the browser. Although a document-dependent optimisation (that take these infinitely many trees into account) seems appropriate, this is far from realistic given the long-standing difficulty of performing sound static analysis for JavaScript especially in the presence of DOM-trees (Andreasen and Møller 2014; Hague et al. 2015; Jensen et al. 2011, 2009; Schäfer et al. 2013; Sridharan et al. 2012). This would make an interesting long-term research direction with many more advances on static analysis for JavaScript. However, the problem is further compounded by the multitude of frameworks deployed on the server-side for HTML creation (e.g. PHP, Java Server Pages, etc.), for which individual tools will need to be developed. Until then, a practical minifier for HTML5 applications will have to make do with document-independent optimisations for CSS files.

The authors of (Mesbah and Mirshokraie 2012) developed a document-dependent dynamic analysis technique for detecting and removing unused CSS selectors in a CSS file that is part of an HTML5 application. A similar tool, called UnCSS (Martino and Contributors 2013), was also later developed. This is done by instrumenting the HTML5 application and removing CSS selectors that have not been used by the end of the instrumentation. The drawback of this technique is that it cannot test all possible behaviours of an HTML5 application and may may accidentally delete selectors that can in reality be used by the application. It was noted in (Hague et al. 2015) that such tools may accidentally delete selectors, wherein the HTML5 application has event listeners that require user interactions. The same paper (Hague et al. 2015) develops a static analysis technique for over-approximating the set of generated DOM-trees by using tree rewriting for abstracting the dynamics of an HTML5 application. The technique, however, covers only a very small subset of JavaScript, and is difficult to extend without first overcoming the hard problem of static analysis of JavaScript.

The authors of (Bosch et al. 2015) applied their earlier techniques (Genevès et al. 2012) to develop a document-independent CSS refactoring/minification technique that removes "redundant" property declarations, and merges two rules with semantically equivalent selectors. The optimisations that they considered are orthogonal to and can be used in conjunction with the optimisation that we consider in this paper. More precisely, they developed an algorithm for checking selector subsumption (given two selectors S_1 and S_2, whether the selector S_1 is subsumed by the selector S_2, written $S_1 \subseteq S_2$). A redundant property declaration p in a rule R_1 with a selector S_1 can, then, be detected by finding a rule R_2 that also contains the declaration p and has a selector S_2 with a higher specificity than S_1 and that $S_1 \subseteq S_2$. As another example, whenever we can show that the selectors S_1 and S_2 of two rules R_1 and R_2 to be semantically equivalent (i.e. $S_1 \subseteq S_2$ and $S_2 \subseteq S_1$), we may merge R_1 with R_2 under certain conditions. The authors of (Bosch et al. 2015) provided sufficient conditions for performing this merge by relating the specificities of S_1 and S_2 with the specificities of other related selectors in the file (but not accounting for the order of appearances of these rules in the file). In general, a CSS rule might have multiple selectors (a.k.a. selector group), each with a different specificity, and it is not clear from the presentation of the paper (Bosch et al. 2015) how their optimisations extend to the general case.

The authors of (Mazinanian et al. 2014) developed a document-dependent[9] CSS refactoring/minification method with an advanced type of rule merging as one of their optimisations. This is an ambitious work utilising a number of techniques from areas such as data mining and constraint satisfaction. Although their work differs from ours because of its document-dependence, the use of refactoring is closely related to our own, hence we will describe in detail some key differences with our approach. The techniques presented in this paper can be viewed as a substantial generalisation of their rule merging optimisation. Loosely speaking, in terms of our graph-theoretic framework, their technique first enumerates all maximal bicliques with at least two selectors. This is done with the help of an association rule mining algorithm (from data mining) with a set of property declarations viewed as an *itemset*. Second, for each such maximal biclique B, a value n is computed that reflects how much saving will be obtained if B could somehow be inserted into the file and every occurrence of each property declaration in B is erased from the rest of the CSS file. Note that n is independent of where B is inserted into the CSS file. Third, for each such maximal biclique B (ranked according to their values in a non-increasing order), a solver for the (finite-domain) constraint satisfaction problem is invoked to check whether B can be placed in the file (with every occurrence of

[9]More precisely, dependent on a given finite set of HTML documents

each property declaration in B is erased from the rest of the CSS file) while preserving the order dependency. If this check fails, the solver will also be invoked to check if one can insert sub-bicliques of B (with a maximal set S of selectors with $|S| \geq 2$) in the file. Possible positions in the file to place each selector of B are encoded as variables constrained by the edge order dependency that is *relativised* to the provided HTML documents. To test whether two edges should be ordered in this relativised edge order, the selectors are not subject to a full intersection test, but instead a *relativised intersection* test that checks whether there is some node in the given finite set of html documents that is matched by both selectors. Their techniques do not work when the HTML documents are not given, which we handle in this paper. Another major difference to our paper is that their algorithm sequentially goes through every maximal biclique B (ranked according to their values) and checks if it can be inserted into the file, which is computationally too prohibitive especially when the (unrelativised) edge order $\prec$ is used. Our algorithm, instead, fully relegates the search of an appropriate B and the position in the file to place it to a highly-optimised Max-SAT solver, which scales to real-world CSS files. In addition, their type of rule merging is also more restricted than ours for two other reasons. Firstly, the new rule inserted into the file has to contain a maximal set of selectors. This prohibits many refactoring opportunities and in fact does not subsume the merging adjacent rule optimisation of `cssnano` (Briggs and Contributors 2015) in general. For example, consider the CSS file

```
.class1 { color:blue }
.class2 { color:blue }
.class3 { color:red }
.class4 { color:blue }
```

Notice that we cannot group together the first, second, and fourth rules since this would change the colour of a node associated with the classes `class2` and `class3`, or with the classes `.class3` and `class4`. On the other hand, the first two rules can be merged resulting in the new file

```
.class1, .class2 { color:blue }
.class3 { color:red }
.class4 { color:blue }
```

However, this is not permitted by their merging rule since `.class1,.class2{color:blue}` does not contain a maximal set of selectors. Secondly, given a maximal biclique B, their merging operation erases *every* occurrence of the declarations of B everywhere else in the file. This further rules out certain refactoring opportunities. For example, consider the CSS file

```
.class1 { color:blue; font-size: large }
.class2 { color:blue; font-size: large }
.class4 { font-size: large }
.class3 { color:red }
.class4 { color:blue }
```

and observe the following maximal biclique in the file.

```
.class1, .class2, .class4 { color:blue; font-size: large }
```

Unfortunately, this is not a valid refactoring using their merging rule since this CSS file is not equivalent to

```
.class1, .class2, .class4 { color:blue; font-size: large }
.class3 { color:red }
```

nor to the following file.

```
.class3 { color:red }
.class1, .class2, .class4 { color:blue; font-size: large }
```

In this paper, we permit duplicate declarations, and would insert this maximal biclique just before the fourth rule in the file (and perform trim) resulting in the following equivalent file.

```
.class1, .class2, .class4 { color:blue; font-size: large }
.class3 { color:red }
.class4 { color: blue }
```

Finally, each maximal biclique in the enumeration of (Mazinanian et al. 2014) does *not* allow two property declarations with the same property name. As we explained in Section 7, CSS rules satisfying this property are rather common since they support fallback options. Handling such bicliques (which we do in this paper) requires extra technicalities, e.g., the notion of orderable bicliques, and adding an order to the declarations in a biclique.

Finally, we mention that there have been works (Hottelier and Bodik 2015; Meyerovich and Bodík 2010; Panchekha and Torlak 2016) on solving web page layout using constraint solvers. These works are orthogonal to this paper. For example, (Panchekha and Torlak 2016) provides a mechanised formalisation of the semantics of CSS for web page layout (in quantifier-free linear arithmetic), which allows them to use an SMT-solver to automatically reason about layout. Our work provides a full formalisation of CSS selectors, which is not especially relevant for layout. Conversely, the layout semantics of various property declarations is not relevant in our rule refactoring problem.

10 CONCLUSION AND FUTURE WORK

We have presented a new technique for minimising CSS stylesheets composed of CSS rules which contain a set of CSS Level 3 selectors and list of property declarations. This technique exploits the fact that new rules may be introduced that render other parts of the document redundant. After removing the redundant parts, the overall file size may be reduced. Such a solution has required the development of a complete formalisation of CSS selectors and their intersection problem as well as a formalisation of the dependency ordering present in a stylesheet. This intersection problem was solved by means of an efficient encoding to quantifier-free integer linear arithmetic, for which there are highly-optimised SMT solvers. Moreover, we have formalised our rule refactoring problem and presented a solution to this problem using an efficient encoding into MaxSAT formulas. These techniques have been implemented in our tool SatCSS which we have comprehensively compared with state-of-the-art minification tools. Our results show clear benefits of our approach.

CSS preprocessors such as Less (Sellier and Contributors 2009) and Sass (Catlin et al. 2006) — which extend the CSS language with useful features such as variables and partial rules — are commonly used in web development. Since Less and Sass code is compiled into CSS before deployment, our techniques are still applicable.

There are many technologies involved in website development and deployment. These technologies provide a variety of options for further research, some of which we briefly discuss here.

Firstly, we may expand the scope of the CSS files we target. For example, we may expand our definition of CSS selectors to include features proposed in the CSS Selectors Level 4 working draft (Etemad and Jr. 2013) (i.e. still not stable). These features include extensions of the negation operator to allow arbitrary selectors to be negated. It would be interesting to investigate the impact of these features on the complexity of the intersection problem.

Another related technology is that of *media queries* that allow portions of a CSS file to only be applied if the host device has certain properties, such as a minimum screen size. This would involve defining semantics of media queries (not part of selectors), and extending our refactoring problem to include refactoring similar media queries and rules grouped under media queries.

Secondly, we could also consider additional techniques for stylesheet optimisation. Currently we take a greedy approach where we search for the "best" refactoring at each iteration. Techniques such as simulated annealing allow a proportion of non-greedy steps to be applied (i.e. choose a refactoring that does not provide the largest reduction in file size). This allows the refactoring process to explore a larger search space, potentially leading to improved final results. Another approach might be to search for multiple simultaneous refactorings.

Finally, our current optimisation metric is the raw file size. We could also attempt to provide an encoding that seeks to find the best file size reduction after gzip compression. [Gzip is now supported by many web hosts and most modern browsers (though not including old IE browsers).] One simple technique that could help bring down the compressed file size is to sort the selectors and declarations in each rule after the minification process is done (Joseph R. Lewis 2010).

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council [EP/K009907/1] and Google Faculty Award. We also thank Davood Mazinanian for answering questions about his work, and Radu Grigore for providing feedback on our paper.

REFERENCES

Alexa Internet. 2017. Alexa Top 500 Global Sites. https://www.alexa.com/topsites. (2017). Referred in April 2017.

E. Andreasen and A. Møller. 2014. Determinacy in static analysis for jQuery. In *OOPSLA*. 17–31. DOI: https://doi.org/10.1145/2660193.2660214

Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. 2016. Max-SAT'16 Competition. http://maxsat.ia.udl.cat/. (2016). Referred in April 2017.

Tab Atkins Jr., Elika J. Etemad, and Florian Rivoal. 2017. CSS Snapshot 2017. https://www.w3.org/TR/css-2017. (2017). Referred in August 2017.

Michael Benedikt, Wenfei Fan, and Floris Geerts. 2005. XPath satisfiability in the presence of DTDs. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*. 25–36. DOI: https://doi.org/10.1145/1065167.1065172

Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. 2003. Structural Properties of XPath Fragments. In *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*. 79–95. http://dx.doi.org/10.1007/3-540-36285-1_6

Nikolaj Bjørner and Nina Narodytska. 2015. Maximum Satisfiability Using Cores and Correction Sets. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. 246–252. http://ijcai.org/Abstract/15/041

Johan Bleuzen. 2017. cssmin. https://www.npmjs.com/package/cssmin. (2017). Referred August 2017.

Mikołaj Bojańczyk. 2010. Automata for Data Words and Data Trees. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*. 1–4. DOI: https://doi.org/10.4230/LIPIcs.RTA.2010.1

Bert Bos. 2016. Cascading Style Sheets Level 2 Revision 2 (CSS 2.2) Specification. https://www.w3.org/TR/CSS22/. (2016). Referred August 2017.

Martí Bosch, Pierre Genevès, and Nabil Layaïda. 2015. Reasoning with Style. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. 2227–2233. http://ijcai.org/Abstract/15/315

Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Ben Briggs and Contributors. 2015. cssnano. http://cssnano.co. (2015). Referred in August 2017.

Hampton Catlin, Natalie Weizenbaum, Chris Eppstein, and Contributors. 2006. Sass. http://sass-lang.com/. (2006). Referred in August 2017.

T. Çelik, E. J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams. 2011. Selectors Level 3: W3C Recommendation 29 September 2011. http://www.w3.org/TR/2011/REC-css3-selectors-20110929/. (2011). Referred in August 2017.

Steve Clay and Contributors. 2017. minify. https://github.com/mrclay/minify. (2017). Referred in August 2017.

Claire David, Leonid Libkin, and Tony Tan. 2012. Efficient reasoning about data trees via integer linear programming. *ACM Trans. Database Syst.* 37, 3 (2012), 19:1–19:28. DOI: https://doi.org/10.1145/2338626.2338632

L. Mendonça de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

Roman Dvornov and Contributors. 2017. csso. https://github.com/css/csso. (2017). Referred in August 2017.

Elika J. Etemad and Tab Atkins Jr. 2013. Selectors Level 4: W3C Working Draft 2 May 2013. http://www.w3.org/TR/2013/WD-selectors4-20130502/. (2013).

Diego Figueira. 2010. *Reasoning on words and trees with data: On decidable automata on data words and data trees in relation to satisfiability of LTL and XPath*. Ph.D. Dissertation. Ecole Normale Superieure de Cachan.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

Floris Geerts and Wenfei Fan. 2005. Satisfiability of XPath Queries with Sibling Axes. In *Database Programming Languages, 10th International Symposium, DBPL 2005, Trondheim, Norway, August 28-29, 2005, Revised Selected Papers*. 122–137. http://dx.doi.org/10.1007/11601524_8

Pierre Genevès and Nabil Layaïda. 2006. A system for the static analysis of XPath. *ACM Trans. Inf. Syst.* 24, 4 (2006), 475–502. DOI: https://doi.org/10.1145/1185882

Pierre Genevès, Nabil Layaïda, and Vincent Quint. 2012. On the analysis of cascading style sheets. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*. 809–818. DOI: https://doi.org/10.1145/2187836.2187946

Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. 2015. Efficiently Deciding μ-Calculus with Converse over Finite Trees. *ACM Trans. Comput. Log.* 16, 2 (2015), 16:1–16:41. DOI: https://doi.org/10.1145/2724712

Georg Gottlob and Christoph Koch. 2002. Monadic Queries over Tree-Structured Data. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 189–202. DOI: https://doi.org/10.1109/LICS.2002.1029828

Matthew Hague, Anthony Widjaja Lin, and C.-H. Luke Ong. 2015. Detecting redundant CSS rules in HTML5 applications: a tree rewriting approach. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 1–19. DOI: https://doi.org/10.1145/2814270.2814288

Jake Hickenlooper. 2014. stripmq. https://www.npmjs.com/package/stripmq. (2014). Referred in August 2017.

Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, and Silvia Pfeiffer. 2014. HTML5. https://www.w3.org/TR/html5/. (2014). Referred August 2017.

Jan Hidders. 2003. Satisfiability of XPath Expressions. In *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*. 21–36. http://dx.doi.org/10.1007/978-3-540-24607-7_3

Thibaud Hottelier and Rastislav Bodik. 2015. Synthesis of Layout Engines from Relational Constraints. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 74–88. DOI: https://doi.org/10.1145/2814270.2814291

S. H. Jensen, M. Madsen, and A. Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *SIGSOFT/FSE*. 59–69. DOI: https://doi.org/10.1145/2025113.2025125

S. H. Jensen, A. Møller, and P. Thiemann. 2009. Type Analysis for JavaScript. In *SAS*. 238–255. DOI: https://doi.org/10.1007/978-3-642-03237-0_17

Meitar Moscovitz Joseph R. Lewis. 2010. *AdvancED CSS*. APress.

Enver Kayaaslan. 2010. On Enumerating All Maximal Bicliques of Bipartite Graphs. In *9th Cologne-Twente Workshop on Graphs and Combinatorial Optimization, Cologne, Germany, May 25-27, 2010. Extended Abstracts*. 105–108.

Leonid Libkin. 2006. Logics for Unranked Trees: An Overview. *Logical Methods in Computer Science* 2, 3 (2006). DOI: https://doi.org/10.2168/LMCS-2(3:2)2006

Leonid Libkin and Cristina Sirangelo. 2010. Reasoning about XML with temporal logics and automata. *J. Applied Logic* 8, 2 (2010), 210–232. DOI: https://doi.org/10.1016/j.jal.2009.09.005

Glacomo Martino and Contributors. 2013. UnCSS. https://github.com/giakki/uncss. (2013). Referred April 2017.

Maarten Marx. 2005. Conditional XPath. *ACM Trans. Database Syst.* 30, 4 (2005), 929–959. DOI: https://doi.org/10.1145/1114244.1114247

Maarten Marx and Maarten de Rijke. 2005. Semantic characterizations of navigational XPath. *SIGMOD Record* 34, 2 (2005), 41–46. DOI: https://doi.org/10.1145/1083784.1083792

Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. 2014. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 496–506. DOI: https://doi.org/10.1145/2635868.2635879

Ali Mesbah and Shabnam Mirshokraie. 2012. Automated analysis of CSS rules to support style maintenance. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 408–418. DOI: https://doi.org/10.1109/ICSE.2012.6227174

Leo A. Meyerovich and Rastislav Bodík. 2010. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*. 711–720. DOI:https://doi.org/10.1145/1772690.1772763

A. Muscholl and I. Walukiewicz. 2005. An NP-complete fragment of LTL. *Int. J. Found. Comput. Sci.* 16, 4 (2005), 743–753. DOI:https://doi.org/10.1142/S0129054105003261

Nina Narodytska and Fahiem Bacchus. 2014. Maximum Satisfiability Using Core-Guided MaxSAT Resolution. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.* 2717–2723. http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8513

Frank Neven. 2002. Automata Theory for XML Researchers. *SIGMOD Rec.* 31, 3 (Sept. 2002), 39–46. DOI:https://doi.org/10.1145/601858.601869

Frank Neven and Thomas Schwentick. 2006. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science* 2, 3 (2006). DOI:https://doi.org/10.2168/LMCS-2(3:1)2006

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, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 181–194. DOI:https://doi.org/10.1145/2983990.2984010

Jakub Pawlowicz. 2017. clean-css. https://github.com/jakubpawlowicz/clean-css. (2017). Referred in August 2017.

René Peeters. 2003. The maximum edge biclique problem is NP-complete. *Discrete Applied Mathematics* 131, 3 (2003), 651–654. DOI:https://doi.org/10.1016/S0166-218X(03)00333-0

Justyna Petke. 2015. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Springer. DOI:https://doi.org/10.1007/978-3-319-21810-6

Armin Rigo and Contributors. 2007. PyPy. http://pypy.org/. (2007). Referred in August 2017.

B. Scarpellini. 1984. Complexity of Subcases of Presburger Arithmetic. *Trans. of AMS* 284, 1 (1984), 203–218.

M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. 2013. Dynamic determinacy analysis. In *PLDI*. 165–174. DOI:https://doi.org/10.1145/2462156.2462168

Helmut Seidl, Thomas Schwentick, Anca Muscholl, and Peter Habermehl. 2004. Counting in Trees for Free. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*. 1136–1149. http://dx.doi.org/10.1007/978-3-540-27836-8_94

Alexis Sellier and Contributors. 2009. Less. http://lesscss.org/. (2009). Referred in August 2017.

Thomas Sha and Contributors. 2014. YUI Compressor. http://yui.github.io/yuicompressor/. (2014). Referred August 2017.

Slant. 2017. Eight Best CSS Minifiers as of 2017. https://www.slant.co/topics/261/~best-css-minifiers. (2017). Referred in April 2017.

Jennifer Slegg. 2017. Google Mobile First Index: Page Speed Included as a Ranking Factor. http://www.thesempost.com/google-mobile-first-index-page-speed-ranking/. *The SEM Post* (23 March 2017).

Steve Souders. 2007. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly Media.

M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*. 435–458. DOI:https://doi.org/10.1007/978-3-642-31057-7_20

Larry J. Stockmeyer and Albert R. Meyer. 1973. Word Problems Requiring Exponential Time: Preliminary Report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*. 1–9. DOI:https://doi.org/10.1145/800125.804029

Balder ten Cate, Tadeusz Litak, and Maarten Marx. 2010. Complete axiomatizations for XPath fragments. *J. Applied Logic* 8, 2 (2010), 153–172. DOI:https://doi.org/10.1016/j.jal.2009.09.002

Balder ten Cate and Maarten Marx. 2007. Navigational XPath: calculus and algebra. *SIGMOD Record* 36, 2 (2007), 19–26. DOI:https://doi.org/10.1145/1328854.1328858

Balder ten Cate and Maarten Marx. 2009. Axiomatizing the Logical Core of XPath 2.0. *Theory Comput. Syst.* 44, 4 (2009), 561–589. DOI:https://doi.org/10.1007/s00224-008-9151-9

Unicode, Inc. 2016. The Unicode Standard, Version 9.0. http://www.unicode.org/versions/Unicode9.0.0. (2016). Referred in August 2017.

Moshe Y. Vardi. 1995. An Automata-Theoretic Approach to Linear Temporal Logic. In *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, August 27 - September 3, 1995, Proceedings)*. 238–266. DOI:https://doi.org/10.1007/3-540-60915-6_6

Mihalis Yannakakis. 1978. Node- and Edge-Deletion NP-Complete Problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*. 253–264. DOI:https://doi.org/10.1145/800133.804355

Appendix

A ADDITIONAL MATERIAL FOR MAX-SAT ENCODING

Recall, given a valid covering $C = \{\overline{B}_i\}_{i=1}^m$ of a CSS graph $\mathcal{G}$, we aim to find a rule $\overline{B} = (X, \overline{Y})$ and a position j that minimises the weight of $C[\overline{B} \to j]_\downarrow$.

In this section we give material omitted from Section 7. We begin with a definition of orderability that will be useful for the remainder of the section. Then we will discuss how to produce the pair $(\{M_i\}_{i=1}^\mu, \mathcal{F})$. Finally we will show that our encoding is correct.

A.1 Orderable Bicliques

To insert a biclique $B = (X, Y)$ into the covering, we need to make sure the order of its edges respects the edge order. We can only order the edges by ordering the properties in the biclique. More precisely, if we insert the biclique at position j, we need all edges in B that do not appear later in the file (i.e. in $\{\overline{B}_i\}_{i=j+1}^m$) to respect the edge order. This is because it is only the last occurrence of an edge that influences the semantics of the stylesheet. Thus, let

$$E_j^B = \{e \in B \mid \text{index}(e) \leq j\} \ .$$

The edge ordering implies a required ordering of E_j^B, which implies an ordering on the properties in Y. This ordering is defined as follows. For all $p_1, p_2 \in Y$ we have

$$p_1 \ll_j^B p_2 \Leftrightarrow \exists (s_1, p_1), (s_2, p_2) \in E_j^B \ . \ (s_1, p_1) \prec^* (s_2, p_2) \ .$$

That is, we require p_1 to appear before p_2 if there are two edges (s_1, p_1) and (s_2, p_2) in B that must be ordered according to the transitive closure of $\prec$. A biclique is orderable iff its properties can be ordered in such a way to respect $\ll_j^B$.

Definition A.1 (Orderable Bicliques). The biclique B is orderable at j if $\ll_j^B$ is acyclic. That is, there does not exist a sequence $(s_1, p_1), \ldots, (s_n, p_n)$ such that $(s_i, p_i) \ll_j^M (s_{i+1}, p_{i+1})$ for all $1 \leq i < n$ and $(s_1, p_1) = (s_n, p_n)$.

This can be easily checked in polynomial time. Moreover, if a biclique is orderable at a given position, a suitable ordering can be found by computing $\ll_j^B$, also in polynomial time. Thus, this proves Proposition 7.1.

A.2 Enumerating Maximal Rules

Fix a covering $C = \{\overline{B}_i\}_{i=1}^m$ of a CSS graph $\mathcal{G} = (S, P, E, \prec, \mathtt{wt})$. We show how to efficiently create the pair $(\{M_i\}_{i=1}^\mu, \mathcal{F})$. In fact, we will create the pair $(\{M_i\}_{i=1}^\mu, \mathcal{F}^{1\text{st}})$ where $\mathcal{F}^{1\text{st}}(j)$ is set of M_i for which j is the smallest position such that M_i is unorderable at position j. Computing $\mathcal{F}$ from this is straightforward (though unnecessary since our encoding uses $\mathcal{F}^{1\text{st}}$ directly).

Our algorithm begins with an enumeration of the maximal bicliques that are orderable at position 0, and iterates up to position m, extending the enumeration at each step and recording in $\mathcal{F}^{1\text{st}}$ which maximal bicliques become unorderable at each position. In fact, we will construct a pair $(\mathcal{M}, \mathcal{F}^{1\text{st}})$ where $\mathcal{M}$ is a set of bicliques rather than a sequence. To construct a sequence we apply any ordering to the elements of $\mathcal{M}$.

A.2.1 Initialisation.

At position 0, all maximal bicliques (X, Y) with $X \times Y \subseteq E$ are orderable. This is because all edges appearing in (X, Y) also appear in $\{\overline{B}_i\}_{i=1}^m$, hence, the semantics of $C[\overline{B} \to 0]$ will be decided by edges already in C, and thus the ordering of the properties does not matter. Hence, we begin with a set of all maximal bicliques $\mathcal{M}_0$. That is, all bicliques (X, Y) such that

$X \times Y \subseteq E$ where there is no (X', Y') with $X' \times Y' \subseteq E$ and $X \times Y \subset X' \times Y'$. Algorithms for generating such an enumeration are known. For example, we use the algorithm from Kayaaslan (Kayaaslan 2010).

For an initial value $\mathcal{F}_0^{1st}$ of $\mathcal{F}^{1st}$, we can simply take the empty function $\emptyset$.

A.2.2 Iteration. Assume we have generated $(\mathcal{M}_j, \mathcal{F}_j^{1st})$. We show how to generate the extension $(\mathcal{M}_{j+1}, \mathcal{F}_{j+1}^{1st})$.

The idea is to find all elements of $\mathcal{M}_j$ that are not orderable at position $j+1$. Let χ be this set. We first define

$$\mathcal{F}_{j+1}^{1st} = \mathcal{F}_j^{1st} \cup \{(j+1, \chi)\} \ .$$

Then, for each biclique $M \in \chi$, we search for smaller bicliques contained within M that are maximal and orderable at position $j+1$. This results in the extension $\{M_i\}_{i=1}^{\mu_{j+1}}$ giving us $(\{M_i\}_{i=1}^{\mu_{j+1}}, \mathcal{F}_{j+1}^{1st})$.

Thus, the problem reduces to finding bicliques contained within some M that are maximal and orderable at position $j+1$. We describe a simple algorithm for this in the next section.

A.2.3 Algorithm for $(\mathcal{M}, \mathcal{F}^{1st})$. We write $\mathrm{Orderable}(M, j)$ to assert that a biclique M is orderable at position j. For now, assume we have the subroutine $\mathrm{OrderableSub}(M, j)$ which returns a set $\mathcal{M}'$ of all orderable maximal bicliques at position j contained within M. We use the following algorithm to generate $(\mathcal{M}, \mathcal{F}^{1st})$.

```
$\mathcal{M} := \mathcal{M}_0$
$\mathcal{F}^{1st} := \emptyset$
for $j := 1$ to $m$ do
    $\chi := \emptyset$
    for all $M \in \mathcal{M}$ do
        if $\neg\mathrm{Orderable}(M, j)$ then
            $\chi := \chi \cup \{M\}$
            $\mathcal{M} := \mathcal{M} \cup \mathrm{OrderableSub}(M, j)$
        end if
    end for
    $\mathcal{F}^{1st} := \mathcal{F}^{1st} \cup \{(j+1, \chi)\}$
end for
return $(\mathcal{M}, \mathcal{F}^{1st})$
```

Note, we can improve the algorithm by restricting the nested for all loop over elements $M \in \mathcal{M}$ to only those M that are not orderable at m. This is because an ordering at position m is also an ordering at position $j <= m$. Hence, these bicliques will never be unorderable and do not need to be checked repeatedly.

A.2.4 Generating Orderable Sub-Bicliques. We now give an algorithm for implementing the subroutine $\mathrm{OrderableSub}(M, j)$. Naively we can simply generate all sub-bicliques M' of M and check $\mathrm{Orderable}(M', j)$. However, to avoid the potentially high cost of such an iteration, we first determine which selectors and properties contribute to $\ll_j^M$. Removing nodes outside of these sets will not affect the orderability, hence we do not need to try removing them. Then we first attempt only removing one node from this set, computing all sub-bicliques that have one fewer element and are orderable. Then, for all nodes for which this fails, we attempt to remove two nodes, and so on. Note, if removing node w renders M orderable, we do not need to test any bicliques obtained by removing w and some other node w', since this will not result in a maximal biclique.

Hence, we define the sets of candidate selectors and properties that may be removed to restore orderability. These are all selectors and nodes that contribute to $\ll_j^M$. That is

$$\Delta = \left\{ s_1, s_2, p_1, p_2 \;\middle|\; \exists (s_1, p_1), (s_2, p_2) \in E_j^M \;.\; (s_1, p_1) \prec^* (s_2, p_2) \right\} \;.$$

We define $\mathrm{OrderableSub}(M, j) = \mathrm{OrderableSub}(M, j, \Delta)$ where $\mathrm{OrderableSub}(M, j, \Delta)$ generates a set Ω of orderable sub-bicliques and is defined below. When $M = (X, Y)$ we will abuse notation and write $M \setminus \{w\}$ for $(X \setminus \{w\}, Y)$ when w is a selector, and $(X, Y \setminus \{w\})$ when w is a property. When defining the algorithm, we will collect all orderable bicliques in a set Ω. We will further collect in Δ' the set of all nodes which fail to create an orderable biclique when removed by themselves. We define $\mathrm{OrderableSub}(M, j, \Delta)$ recursively, where the recursive call attempts the removal of an increasing number of nodes. It is

```
$\Omega := \emptyset$
$\Delta' := \emptyset$
**for all** $w \in \Delta$ **do**
    $M' := M \setminus \{w\}$
    **if** $\mathrm{Orderable}(M', j)$ **then**
        $\Omega := \Omega \cup \{M'\}$
    **else**
        $\Delta' := \Delta' \cup \{w\}$
    **end if**
**end for**
**for all** $w \in \Delta'$ **do**
    $\Omega := \Omega \cup \mathrm{OrderableSub}(M \setminus \{w\}, j, \Delta' \setminus \{w\})$
**end for**
**return** $\Omega$
```

A.3 Correctness of the Encoding

We argue Proposition 7.3 which claims that the encoding (Π_H, Π_S) is correct. To prove this we need to establish three facts.

(1) If $(\overline{B}, j)$ is a valid refactoring of C and $\overline{B} = (B, \lhd)$, then (B, j) is a solution to the hard constraints.
(2) If $(\overline{B}, j)$ is generated from a solution to the hard constraints, it is a valid refactoring.
(3) The weight of a solution generating $(\overline{B}, j)$ is the size of $C[\overline{B} \to j]_{\downarrow}$.

We argue these properties below.

(1) Take a valid refactoring $(\overline{B}, j)$ and let $\overline{B} = (B, \lhd)$. We construct a solution to the hard constraints. First, we assign $\bar{j} = j$. Next, since the refactoring is valid, we know B contains only edges in E. That is, it is contained within a maximal biclique. Furthermore, since $C[\overline{B} \to j]$ is valid, we know that B is orderable at position j. Thus, it is a sub biclique of some M_i in $(\{M_i\}_{i=1}^{\mu}, \mathcal{F}^{1\text{st}})$, and, moreover, it is not the case that $M_i \in \mathcal{F}^{1\text{st}}(j')$ for some $j' \leq j$. Thus, we assign $\bar{\imath}_M = i$ and we know that

$$\bigwedge_{1 \leq j \leq m+1} \left(\left(\bar{j} >= j\right) \Rightarrow \bigwedge_{M_i \in \mathcal{F}^{1\text{st}}(j)} \left(\bar{\imath}_M \neq i\right) \right)$$

is satisfied.

Additionally, for all w appearing in B but not in M_i, we set $\overline{x}_{\rho_i(w)}$ to false, otherwise we set it to true. Thus $\mathrm{HasEdge}((s, p))$ holds only if (s, p) is an edge in B.

Next, we argue

$$\left(\bigwedge_{(s_1,p_1) \prec (s_2,p_2)} \begin{array}{l} \text{HasEdge}((s_1,p_1)) \Rightarrow \\ \quad \left(\bar{j} \leq \text{index}((s_2,p_2)) \vee \text{HasEdge}((s_2,p_2))\right) \end{array} \right)$$

is satisfied. This follows from $C[\overline{B} \to j]$ being valid. To see this, take some $(s_1,p_1) \prec (s_2,p_2)$. If (s_1,p_1) does not appear in B, then there is nothing to prove. If it does, we know (s_2,p_2) must appear later in the file. There are two cases. If $j \leq \text{index}((s_2,p_2))$ then the clause is satisfied. Otherwise we must have (s_2,p_2) in B or edge order would be violated. Thus the clause also holds in this case.

(2) We need to prove that if the hard constraints are satisfied, then then generated refactoring $(\overline{B}, j)$ is valid. Let $\overline{B} = (B, \lhd)$. For $C[\overline{B} \to j]$ to be valid, we first have to show that B introduces no new edges to the stylesheet. This is immediate since B is a sub biclique of some M_i in $(\{M_i\}_{i=1}^{\mu}, \mathcal{F}^{1\text{st}})$, which can only contain edges in E.

Next, we need to argue that we can create the ordering $\lhd$ for the properties in B. First note that M_i is orderable at position j. In particular, for any $(s_1,p_1) \prec^* (s_2,p_2)$ with (s_1,p_1) and (s_2,p_2) appearing in M_i, we have $p_1 \ll_j^{M_i} p_2$. Since all edges in B also appear in M_i, the existence of an ordering is immediate.

Finally, we need to argue that $C[\overline{B} \to j]$ respects the edge order. Suppose $(s_1,p_2) \prec (s_2,p_2)$. To violate this ordering, we need to introduce a copy of (s_1,p_1) after the last copy of (s_2,p_2). Thus, we must have (s_1,p_1) in B. However, from

$$\left(\bigwedge_{(s_1,p_1) \prec (s_2,p_2)} \begin{array}{l} \text{HasEdge}((s_1,p_1)) \Rightarrow \\ \quad \left(\bar{j} \leq \text{index}((s_2,p_2)) \vee \text{HasEdge}((s_2,p_2))\right) \end{array} \right)$$

we are left with two cases. In the first $j \leq \text{index}((s_2,p_2))$ and the edge order is maintained. In the second, we also have (s_2,p_2) in B. However, the edge order is maintained because B is orderable. Thus we are done.

(3) Finally, we argue that the weight of a satisfying assignment accurately reflects the size of $C[\overline{B} \to j]_{\downarrow}$. This is fairly straightforward. The size of $C[\overline{B} \to j]_{\downarrow}$. comprises two parts: the size of $\overline{B}$, and the size of C after the trim operation. It is immediate to see that the size of $\overline{B}$ is equal to the size of all of its nodes. In particular, this is the size of all nodes of M_i that appear in $\overline{B}$. That is, have not been excluded. Thus the clause with weight $\mathtt{wt}(w)$

$$\left(\bar{i}_M = i\right) \Rightarrow \overline{x}_{\rho_i(w)} \ .$$

for each w appearing in M_i accurately computes the size of $\overline{B}$.

For the size of C after the trim operation, we first use the assumption that C has already been trimmed before the refactoring. Thus, any further nodes removed in $C[\overline{B} \to j]_{\downarrow}$ from a rule $\overline{B}_{i'}$ must be removed because some edge e in $\overline{B}$ also appears in $\overline{B}_{i'}$ and, moreover, it was the case $i' = \text{index}(e)$ and $i' \leq j$. In particular, we can only remove a node w from $\overline{B}_{i'}$ if all edges e incident to w with $i' = \text{index}(e)$ have e appearing in $\overline{B}$ (else there will still be some edge preventing w from being trimmed after the refactoring). Thus, for each selector node s, we know it is not removed if the clause with weigth $\mathtt{wt}(s)$

$$i \leq \bar{j} \wedge \bigwedge_{\substack{\text{index}((s,p))=i \\ p \in \overline{Y}}} \text{HasEdge}((s,p))$$

is not satisfied. Similarly for property nodes p. Thus, these clauses accurately count the size of the covering after trimming.

B ADDITIONAL MATERIAL FOR SECTION 5

B.1 Handling Pseudo-Elements

CSS selectors can also finish with a *pseudo-element*. For example φ `::before`. These match nodes that are not formally part of a document tree. In the case of φ `::before` the selector matches a phantom node appearing before the node matched by φ. These can be used to insert content into the tree for stylistic purposes. For example

```
.a::before { content:">" }
```

places a ">" symbol before the rendering of any node with class `a`.

We divide CSS selectors into five different types depending on the pseudo-element appearing at the end of the selector. We are interested here in the nodes matched by a selector. The pseudo-elements `::first-line`, `::first-letter`, `::before`, and `::after` essentially match nodes inserted into the DOM tree. The CCS3 specification outlines how these nodes should be created. For our purposes we only need to know that the five syntactic cases in the above grammar can never match the same inserted node, and the selectors `::first-letter` and `::first-line` require that the node matched by φ is not empty.

Since we are interested here in the non-emptiness and non-emptiness-of-intersection problems, we will omit pseudo-elements in the sequel, under the assumptions that

- selectors of the form φ `::first-line` or φ `::first-letter` are replaced by a selector φ `:not(:empty)`, and
- selectors of the form φ, φ `::before`, or φ `::after` are replaced by φ, and
- we *never* take the intersection of two selectors φ and φ' such that it's not the case that either
  - φ and φ' were derived from selectors containing no pseudo-elements, or
  - φ and φ' were derived from selectors ending with the same pseudo-element.

In this way, we can test non-emptiness of a selector by testing its replacement. For non-emptiness-of-intersection, we know if two selectors end with different pseudo-elements (or one does not contain a pseudo-element, and one does), their intersection is necessarily empty. Thus, to check non-emptiness-of-intersection, we immediately return "empty" for any two selectors ending with different pseudo-elements. To check two selectors ending with the same pseudo-element, the problem reduces to testing the intersection of their replacements.

B.2 NP-hardness of Theorem 5.1

LEMMA B.1. *Given a CSS selector φ, deciding $\exists T, \eta \,.\, T, \eta \models \varphi$ is NP-hard.*

PROOF. We give a polynomial-time reduction from the NP-complete problem of non-universality of unions of arithmetic progressions (Stockmeyer and Meyer 1973, Proof of Theorem 6.1). To define this, we first fix some notation. Given a pair $(\alpha, \beta) \in \mathbb{N} \times \mathbb{N}$, we define $[\![(\alpha, \beta)]\!]$ to be the set of natural numbers of the form $\alpha n + \beta$ for $n \in \mathbb{N}$. That is, $[\![(\alpha, \beta)]\!]$ represents an arithmetic progression, where α represents the *period* and β represents the *offset*. Let $E \subseteq \mathbb{N} \times \mathbb{N}$ be a finite subset of pairs (α, β). We define $[\![E]\!] = \bigcup_{(\alpha,\beta)\in E} [\![(\alpha, \beta)]\!]$. The NP-complete problem is: given E (where numbers may be represented in unary or in binary representation), is $[\![E]\!] \neq \mathbb{N}$? Observe that this problem is equivalent to checking whether $[\![E + 1]\!] \neq \mathbb{N}_{>0}$ where $E + 1$ is defined by adding 1 to the offset β of each arithmetic progression (α, β) in E. By complementation, this last problem is equivalent to checking whether $\mathbb{N}_{>0} \setminus [\![E + 1]\!] \neq \emptyset$. Since $\mathbb{N}_{>0} \setminus [\![E + 1]\!] = \bigcap_{(\alpha,\beta)\in E} \overline{[\![\alpha, \beta + 1]\!]}$, the problem can be seen to be equivalent to testing the non-emptiness of

$$\star\{\texttt{:not(:nth-child(}\alpha\texttt{n + }(\beta+1)\texttt{))} \mid (\alpha,\beta) \in E\} \ .$$

Thus, non-emptiness is NP-hard. □

B.3 Handling `!important` and shorthand property names

B.3.1 The !important Keyword. Firstly, the keyword `!important` in property declaration as is used in the rule

```
div { color:red !important }
```

can be used to override the cascading behaviour of CSS, e.g., in our example, if a node is matched by `div`, as well as a later rule R that assigns a different color, then assign `red` to color (unless R also has the keyword `!important` next to its color property declaration). To handle this, we can extend the notion of specificity of a selector to the notion of specificity of a pair (s, p) of selector and property declaration, after which we may proceed as before (i.e. relating only two edges with the same specificity). Recall from (Çelik et al. 2011) that the specificity of a selector is a 3-tuple $(a, b, c) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ where a, b, and c can be obtained by calculating the sum of the number of IDs, classes, tag names, etc. in the selector. Since the lexicographic order is used to rank the elements of $S := \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, the specificity of a pair (s, p) can now be defined to be (i, a, b, c), where (a, b, c) is the specificity of s, and $i = 1$ if `!important` can be found in p (otherwise, $i = 0$). In particular, this also handles the case where multiple occurrences of `!important` is found in the CSS file.

B.3.2 Shorthand Property Names. *Shorthand property names* (Bos 2016) can be used to simultaneously set the values of related property names. For example, `border: 5px solid red` is equivalent to

```
border-width: 5px; border-style: solid; border-color:red
```

In particular, this implies that (s, p) and (s, p') can be related in $\prec$ if p defines `border`, while the other property p' defines `border-width`. One way to achieve this is to simply list all pairs of comparable property names, which can be done since only around 100 property names are currently officially related. [Incidentally, a close enough approximation is that one property name is a *prefix* of the other property name (e.g., `border` is a prefix of `border-style`), but this is not complete (e.g. `font` can be used to define `line-height`)]

C ADDITIONAL MATERIAL FOR SECTION 6

C.1 Correctness of $\mathcal{A}_\varphi$ in Proposition 6.1

We show both soundness and completeness of $\mathcal{A}_\varphi$.

LEMMA C.1. *For each CSS selector φ and tree T, we have*

$$(T, \eta) \in \mathcal{L}\left(\mathcal{A}_\varphi\right) \Rightarrow T, \eta \models \varphi \ .$$

PROOF. Suppose $(T, \eta) \in \mathcal{L}\left(\mathcal{A}_\varphi\right)$. By construction of $\mathcal{A}_\varphi$ we know that the accepting run must pass through all states $\circ_1, \ldots, \circ_n$ where $\varphi = \sigma_1\ o_1\ \cdots\ o_{n-1}\ \sigma_n$. Notice, in order to exit each state $\circ_i$ a transition labelled by σ_i must be taken. Let η_i we the node read by this transition, which necessarily satisfies σ_i. Observe $\eta_n = \eta$. We proceed by induction. We have η_1 satisfies σ_1. Hence, assume η_i satisfies $\sigma_1\ o_1\ \cdots\ o_{i-1}\ \sigma_i$. We show η_{i+1} satisfies $\sigma_1\ o_1\ \cdots\ o_i\ \sigma_{i+1}$.

We case split on o_i.

- When $o_i = \gg$ we need to show η_{i+1} is a descendant of η_i. By construction of $\mathcal{A}_\varphi$ the run reaches η_{i+1} in one of two ways. If it is via a single transition $\circ_i \xrightarrow[\sigma_i]{\downarrow} \circ_{i+1}$ then η_{i+1}

is immediately a descendant of η_i. Otherwise the first transition is $\circ_i \xrightarrow[\sigma_i]{\downarrow} \bullet_i$. The reached node is necessarily a descendant of η_i. To reach η_{i+1} a path is followed applying $\rightarrow_+$ and $\downarrow$ arbitrarily, which cannot reach a node that is not a descendant of η_i. Finally, the transition to η_{i+1} is via $\rightarrow$ or $\downarrow$ and hence η_{i+1} must also be a descendant of η_i.

- When $o_i = >$ we need to show η_{i+1} is a descendant of η_i. By construction of $\mathcal{A}_\varphi$ the run reaches η_{i+1} in one of two ways. If it is via a single transition $\circ_i \xrightarrow[\sigma_i]{\downarrow} \circ_{i+1}$ then η_{i+1} is immediately a child of η_i. Otherwise the first transition is $\circ_i \xrightarrow[\sigma_i]{\downarrow} \bullet_i$. The reached node is necessarily a child of η_i. To reach η_{i+1} only transitions labelled $\rightarrow_+$ and $\rightarrow$ can be followed. Hence, the node reached must also be a child of η_i.
- When $o_i = +$ we need to show η_{i+1} is the next neighbour of η_i. Since the only path is a single transition labelled $\rightarrow$ the result is immediate.
- When $o_i = \sim$ we need to show η_{i+1} is a sibling of η_i. By construction of $\mathcal{A}_\varphi$ the run reaches η_{i+1} in one of two ways. If it is via a single transition $\circ_i \xrightarrow[\sigma_i]{\rightarrow} \circ_{i+1}$ then η_{i+1} is immediately a sibling of η_i. Otherwise the first transition is $\circ_i \xrightarrow[\sigma_i]{\rightarrow} \bullet_i$. The reached node is necessarily a sibling of η_i. To reach η_{i+1} only transitions labelled $\rightarrow_+$ and $\rightarrow$ can be followed. Hence, the node reached must also be a sibling of η_i.

Thus, by induction, $\eta_n = \eta$ satisfies $\sigma_1\ o_1\ \cdots\ o_{n-1}\ \sigma_n = \varphi$. □

LEMMA C.2. *For each CSS selector φ and tree T, we have*

$$T, \eta \models \varphi \Rightarrow (T, \eta) \in \mathcal{L}\left(\mathcal{A}_\varphi\right)$$

PROOF. Assume $T, \eta \models \varphi$. Thus, since $\varphi = \sigma_1\ o_1\ \cdots\ o_{n-1}\ \sigma_n$, we have a sequence of nodes $\eta_1, \ldots, \eta_n$ such that for each i we have $T, \eta_i \models \sigma_1\ o_1\ \cdots\ o_{i-1}\ \sigma_i$. Note $\eta_n = \eta$. We build a run of $\mathcal{A}_\varphi$ from σ_1 to $\circ_i$ by induction. When $i = 1$ we have the run constructed by taking the loops on the initial state $\circ_1$ labelled $\downarrow$ and $\rightarrow_+$ to navigate to η_1. Assume we have a run to $\circ_i$. We build a run to $\circ_{i+1}$ we consider o_i.

- When $o_i = \gg$ we know η_{i+1} is a descendant of η_i. We consider the construction of $\mathcal{A}_\varphi$. If η_{i+1} is the first child of η_i, we construct the run to $\circ_{i+1}$ via the transition $\circ_i \xrightarrow[\sigma_i]{\downarrow} \circ_{i+1}$, noting that we know η_i satisfies σ_i. Otherwise we take $\circ_i \xrightarrow[\sigma_i]{\downarrow} \bullet_i$ and arrive at either an ancestor or sibling of η_{i+1}. In the case of a neighbour, we can take the transition labelled $\rightarrow$ to reach η_{i+1}. For an indirect sibling we can take the transition labelled $\rightarrow_+$ followed by the transition labelled $\rightarrow$. For an ancestor, we take the transition labelled $\downarrow$ and arrive at another sibling or ancestor of η_{i+1} that is closer. We continue in this way until we reach η_{i+1} as needed.
- When $o_i = >$ we know η_{i+1} is a child of η_i. We consider the construction of $\mathcal{A}_\varphi$. If η_{i+1} is the first child of η_i, we construct the run to $\circ_{i+1}$ via the transition $\circ_i \xrightarrow[\sigma_i]{\downarrow} \circ_{i+1}$, noting that we know η_i satisfies σ_i. Otherwise we take $\circ_i \xrightarrow[\sigma_i]{\downarrow} \bullet_i$ and arrive at a preceding sibling of η_{i+1}. We can take the transition labelled $\rightarrow_+$ to reach the preceding neighbour of η_{i+1} if required, and then the transition labelled $\rightarrow$ to reach η_{i+1} as required.

- When $o_i = +$ we know η_{i+1} is the neighbour of η_i. We consider the construction of $\mathcal{A}_\varphi$ and take the only available transition $\circ_i \xrightarrow[\sigma_i]{\rightarrow} \circ_{i+1}$, noting that we know η_i satisfies σ_i. Thus, we reach η_{i+1} as required.
- When $o_i = \sim$ we know η_{i+1} is a sibling of η_i. We consider the construction of $\mathcal{A}_\varphi$. If η_{i+1} is the neighbour of η_i, we construct the run to $\circ_{i+1}$ via the transition $\circ_i \xrightarrow[\sigma_i]{\downarrow} \circ_{i+1}$, noting that we know η_i satisfies σ_i. Otherwise we take $\circ_i \xrightarrow[\sigma_i]{\rightarrow} \bullet_i$ and arrive at a preceding sibling of η_{i+1}. We can take the transition labelled $\rightarrow_+$ to reach the preceding neighbour of η_{i+1} is required, and then the transition labelled $\rightarrow$ to reach η_{i+1} as required.

Thus, by induction, we construct a run to η_n ending in state $\circ_n$. We transform this to an accepting run by taking the transition $\circ_n \xrightarrow[\sigma_n]{\circ} q_f$, using the fact that η_n satisfies σ_n. □

C.2 Proof of Proposition 6.2

We show that

$$(T, \eta) \in \mathcal{L}(\mathcal{A}_1) \wedge (T, \eta) \in \mathcal{L}(\mathcal{A}_2) \Leftrightarrow (T, \eta) \in \mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2) .$$

We begin by observing that that all runs of a CSS automaton showing acceptance of a node η in T must follow a sequence of nodes $\eta_1, \ldots, \eta_n$ such that

- η_1 is the root of T, and
- when $\eta_j = \eta' \iota$ then either $\eta_{j+1} = \eta'(\iota + 1)$ or $\eta_{j+1} = \eta_j 1$ for all j, and
- $\eta_n = \eta$

that defines the path taken by the automaton. Each node is "read" by some transition on each run. Note a transition labelled $\rightarrow_+$ may read sequence nodes that is a factor of the path above. However, since these transitions are loops that do not check the nodes, without loss of generality we can assume each $\rightarrow_+$ in fact reads only a single node. That is, $\rightarrow_+$ behaves like $\rightarrow$. Recall, $\rightarrow_+$ was only introduced to ensure the existence of "short" runs.

Because of the above, any two runs accepting η in T must follow the same sequence of nodes and be of the same length.

We have $(T, \eta) \in \mathcal{L}(\mathcal{A}_1) \wedge (T, \eta) \in \mathcal{L}(\mathcal{A}_2)$ iff there are accepting runs

$$q^i_1 \xrightarrow[\sigma^i_1]{d^i_1} \cdots \xrightarrow[\sigma^i_n]{d^i_n} q^i_{n+1}$$

of $\mathcal{A}_i$ over T reaching node η for both $i \in \{1, 2\}$. We argue these two runs exist iff we have a run

$$\left(q^1_1, q^2_1\right) \xrightarrow[\sigma_1]{d_1} \cdots \xrightarrow[\sigma_n]{d_n} \left(q^1_{n+1}, q^2_{n+1}\right)$$

of $\mathcal{A}_1 \cap \mathcal{A}_2$ where each d_j and σ_j depends on $\left(d^1_j, d^2_j\right)$.

- When $(\downarrow, \downarrow)$ we have $d_j = \downarrow$ and $\sigma_j = \sigma^1_j \cap \sigma^2_j$.
- When $(\rightarrow, \rightarrow)$ we have $d_j = \rightarrow$ and $\sigma_j = \sigma^1_j \cap \sigma^2_j$.
- When $(\rightarrow, \rightarrow_+)$ we have $d_j = \rightarrow$ and $\sigma_j = \sigma^1_j$.
- When $(\rightarrow_+, \rightarrow)$ we have $d_j = \rightarrow$ and $\sigma_j = \sigma^2_j$.
- When $(\rightarrow_+, \rightarrow_+)$ we have $d_j = \rightarrow_+$ and $\sigma_j = \star$.
- When $(\circ, \circ)$ we have $d_j = \circ$ and $\sigma_j = \sigma^1_j \cap \sigma^2_j$.
- The cases $(\downarrow, \rightarrow_+)$, $(\downarrow, \rightarrow)$, $(\downarrow, \circ)$, $(\rightarrow, \downarrow)$, $(\rightarrow, \circ)$, $(\rightarrow_+, \downarrow)$, $(\rightarrow_+, \circ)$, $(\circ, \downarrow)$, $(\circ, \rightarrow)$, and $(\circ, \rightarrow_+)$ are not possible.

The existence of the transitions comes from the definition of $\mathcal{A}_1 \cap \mathcal{A}_2$. We have to argue that η_j satisfies both σ^i_j iff it also satisfies σ_j. By observing $\sigma \cap \star = \star \cap \sigma = \sigma$ we always have $\sigma_j = \sigma^1_j \cap \sigma^2_j$.

Let $\sigma^i_j = \tau_i \Theta_i$ and $\sigma_j = \tau\Theta$. It is immediate that η_j satisfies $\Theta = \Theta_1 \cup \Theta_2$ iff it satisfies both Θ_i.

To complete the proof we need to show η_j satisfies τ iff it satisfies both τ_i. Note, we must have some s and e such that $\tau, \tau_1, \tau_2 \in \{\star, \texttt{(}s \,|\, \star\texttt{)}, \texttt{(}s \,|\, e\texttt{)}, e\}$ else the type selectors cannot be satisfied (either $\tau = \texttt{:not(}\star\texttt{)}$ or τ_1 and τ_2 assert conflicting namespaces or elements).

If some $\tau_i = \star$ the property follows by definition. Otherwise, if $\tau = \tau_2$ then in all cases the conjunction of τ_1 and τ_2 is equivalent to τ_2 and we are done. The situation is similar when $\tau = \tau_1$. Otherwise $\tau = \texttt{(}s \,|\, e\texttt{)}$ and $\tau_1 = \texttt{(}s \,|\, \star\texttt{)}$ and $\tau_2 = e$ or vice versa, and it is easy to see τ is equivalent to the intersection of τ_1 and τ_2. Thus, we are done.

C.3 Proofs for Non-Emptiness of CSS Automata

C.3.1 Bounding Namespaces and Elements. We show Proposition 6.4 (Bounded Types). We need to define the finite sets $\downharpoonleft(E)$ and $\downharpoonleft(s)$. To this end, we write

(1) $E_{\mathcal{A}}$ to denote the set of namespaced elements $s{:}e$ such that there is some transition $q \xrightarrow[\sigma]{d} q' \in \Delta$ with $\sigma = \texttt{(}s \,|\, e\texttt{)}\,\Theta$ for some s, e, and Θ,

(2) $S_{\mathcal{A}}$ is the set of transitions $q \xrightarrow[\sigma]{d} q' \in \Delta$ with $\sigma \neq \star$ and $|\mathcal{A}|_\sigma$ denotes the cardinality of $S_{\mathcal{A}}$.

Let $\left\{\tau_1, \ldots, \tau_{|\mathcal{A}|_\sigma}\right\}$ be a set of fresh namespaced elements and

$$\downharpoonleft(E_{\mathcal{A}}) = E_{\mathcal{A}} \uplus \left\{\tau_1, \ldots, \tau_{|\mathcal{A}|_\sigma}\right\} \uplus \{\bot\}$$

where there is a bijection $\theta : S_{\mathcal{A}} \to \left\{\tau_1, \ldots, \tau_{|\mathcal{A}|_\sigma}\right\}$ such that for each $t \in S_{\mathcal{A}}$ we have $\theta(t) = \tau$ and

(1) $\tau = s{:}e$ if σ can only match elements $s{:}e$,
(2) $\tau = s{:}e$ for some fresh element e if σ can only match elements of the form $s{:}e'$ for all elements e', and
(3) $\tau = s{:}e$ for some fresh namespace s if σ can only match elements of the form $s'{:}e$ for all namespaces s', and
(4) $\tau = s{:}e$ for fresh s and fresh e if σ places no restrictions on the element type.

Thus, we can define bounded sets of namespaces and elements

$$\begin{array}{rcl} \downharpoonleft(E) & = & \{e \mid \exists s \,.\, s{:}e \in \downharpoonleft(E_{\mathcal{A}})\} \\ \downharpoonleft(\mathrm{NS}) & = & \{s \mid \exists e \,.\, s{:}e \in \downharpoonleft(E_{\mathcal{A}})\}\ . \end{array}$$

It remains to show $\downharpoonleft(E_{\mathcal{A}})$ is sufficient. That is, if some tree T is accepted $\mathcal{A}$, we can define another tree T' that also is accepted by $\mathcal{A}$ but only uses types in $\downharpoonleft(E_{\mathcal{A}})$.

We take $(T, \eta) \in \mathcal{L}(\mathcal{A})$ with $T = (D, \lambda)$ and we define $T' = (D, \lambda')$ satisfying the proposition. Let

$$q_0, \eta_0, q_1, \eta_1, \ldots, q_\ell, \eta_\ell, q_{\ell+1}$$

be the accepting run of $\mathcal{A}$, by the sequence of transitions $t_0, \ldots, t_\ell$. As noted above, we can assume each transition in Δ appears only once in this sequence. Let $\left\{\sigma_1, \ldots, \sigma_{|\mathcal{A}|_\sigma}\right\}$ be the set of selectors appearing in $\mathcal{A}$. We perform the following modifications to λ to obtain λ'.

We obtain λ' from λ by changing the element labelling. We first consider all $0 \leq i \leq \ell$ such that η_i is labelled by some element $s{:}e \in E_{\mathcal{A}}$. Let $\mathrm{Nodes}_{s:e}$ be the set of nodes labelled by $s{:}e$ in λ. In λ' we label all nodes in $\mathrm{Nodes}_{s:e}$ by $s{:}e$. That is, we do not relabel nodes labelled by $s{:}e$. Let Nodes be the union of all such $\mathrm{Nodes}_{s:e}$.

Next we consider all $0 \leq i \leq \ell$ such that $\eta_i \notin \mathrm{Nodes}$ (i.e. was not labelled in the previous case) and $t_i = q_i \xrightarrow[\sigma]{d} q_{i+1}$ with $\sigma \neq \star$. Let $s{:}e \notin E_{\mathcal{A}}$ be the element labelling of η_i in λ. Moreover, take τ such that $\theta(t_i) = \tau$. In λ' we label all nodes in $\mathrm{Nodes}_{s:e}$ (i.e. labelled by $s{:}e$) in λ by τ. That is, we globally replace $s{:}e$ by τ. Let Nodes' be Nodes union all such Nodes_e.

Finally, we label all nodes not in Nodes' with the null element $\perp$.

To see that

$$q_0, \eta_0, q_1, \eta_1, \ldots, q_\ell, \eta_\ell, q_{\ell+1}$$

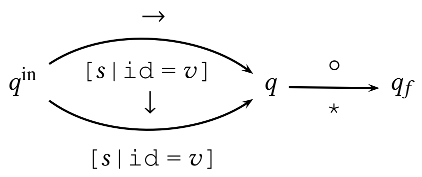
via $t_0, \ldots, t_\ell$ is an accepting run of (D, λ') we only need to show that for each $t_i = q_i \xrightarrow[\sigma]{d} q_{i+1}$ that η_i satisfies σ. This can be shown by induction over σ. Most atomic cases are straightforward (e.g. the truth of `:hover` is not affected by our transformations). The case of e, $(s \mid \star)$, or $(s \mid e)$ appearing positively follows since in these cases the labelling remained the same or changed to some τ consistent with the selector. When such selectors appear negatively, the result follows since we only changed elements and namespaces to fresh ones. The truth of attribute selectors remains unchanged since we did not change the attribute labelling. The cases of `:nth-child(`α`n + `β`)` and `:nth-last-child(`α`n + `β`)` follow since we did not change the number of nodes. For the selectors `:nth-of-type(`α`n + `β`)` and `:nth-last-of-type(`α`n + `β`)` there are two cases. If we did not change the element label $s{:}e$ of η_i, then we also did not change the label of its siblings. Moreover, we did not add any $s{:}e$ labels elsewhere in the tree. Hence the truth of the formulas remains the same. If we did change the label from $s{:}e$ to τ for some τ then observe that we also relabelled all other nodes in the tree labelled by $s{:}e$. In particular, all siblings of η_i. Moreover, since θ is a bijection and each transition appears only once in the run, we did not label any node not labelled $s{:}e$ with τ. Hence the truth of the formulas also remains the same. Similar arguments hold for `:only-child` and `:only-of-type`.

Thus, (D, λ') is accepted, and only uses elements in $\downharpoonleft(E_{\mathcal{A}})$ as required.

C.3.2 Proof of Polynomial Bound on Attribute Value Lengths. We prove Proposition 6.5 (Bounded Attributes). That is we argue the existence of a polynomial bound for the solutions to any finite set C of constraints of the form $[s \mid a \mathrel{\mathrm{op}} v]$ or `:not([s|a op v])`, for some fixed s and a. We say that C is a set of constraints over s and a.

In fact, the situation is a little more complicated because it may be the case that a is `id`. In this case we need to be able to enforce a global uniqueness constraint on the attribute values. Thus, for constraints on an ID attribute, we need a bound that is large enough to allow to all constraints on the same ID appearing throughout the automaton to be satisfied by unique values. Thus, for a given automaton, we might ask for a bound N such that *if* there exists unique ID values for each transition, then there exist values of length bounded by N.

However, the bound on the length must still work when we account for the fact that not all transitions in the automaton will be used during a run. Consider the following illustrative example.


$$\begin{tikzcd} q^{\mathrm{in}} \arrow[r, bend left, "\rightarrow", "{[s \mid \mathtt{id} = v]}"'] \arrow[r, bend right, "\downarrow", "{[s \mid \mathtt{id} = v]}"'] & q \arrow[r, "\circ", "\star"'] & q_f \end{tikzcd}$$

In this case we have two transitions with ID constraints, and hence two sets of constraints $C_1 = C_2 = \{\, \texttt{[} s \,|\, \texttt{id} = v \texttt{]} \,\}$. Since these two sets of constraints cannot be satisfied simultaneously with unique values, even the bound $N = 0$ will satisfy our naive formulation of the required property (since the property had the existence of a solution as an antecedent). However, it is easy to see that any run of the automaton does not use both sets of constraints, and that the bound $N = |v|$ should suffice. Hence, we formulate the property of our bound to hold for all *sub-collections* of the collection of sets of constraints appearing in the automaton.

LEMMA C.3 (BOUNDED ATTRIBUTE VALUES). *Given a collection of constraints $C_1, \ldots, C_n$ over some s and a, there exists a bound N polynomial in the size of $C_1, \ldots, C_n$ such that for any subsequence $C_{i_1}, \ldots, C_{i_m}$ if there is a sequence of words $v_1, \ldots, v_m$ such that all v_j are unique and v_j satisfies the constraints in C_{i_j}, then there is a sequence of words such that the length of each v_j is bounded by N, all v_j are unique, and v_j satisfies the constraints in C_{i_j},*

The proof uses ideas from Muscholl and Walukiewicz's NP fragment of LTL (Muscholl and Walukiewicz 2005). We first, for each set of constraints C, construct a deterministic finite word automaton $\mathcal{A}$ that accepts only words satisfying all constraints in C. This automaton has a polynomial number of states and can easily be seen to have a short solution by a standard pumping argument. Given automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ with at most N_s states and N_c constraints in each set of constraints, we can again use pumping to show there is a sequence of distinct words $v_1, \ldots, v_n$ such that each v_i is accepted by $\mathcal{A}_i$ and the length of v_i is at most $n \cdot N_s \cdot N_c$.

The Automata. We define a type of word automata based on a model by Muscholl and Walukiewicz to show and NP upper bound for a variant of LTL. These automata read words and keep track of which constraints in C have been satisfied or violated. They accept once all positive constraints have been satisfied and no negative constraints have been observed.

In the following, let $\mathrm{Prefs}(C)$ be the set of words v' such that v' is a prefix of some v with $\texttt{[} s \,|\, a \ \mathrm{op}\ v \texttt{]} \in C$ or $\texttt{:not(} \texttt{[s|a op v]} \texttt{)} \in C$. Moreover, let $\hat{}$ and $\$$ be characters not in Γ that will mark the beginning and end of the word respectively. Additionally, let ε denote the empty word. Finally, we write $v \preceq v'$ if v is a *factor* of v', i.e., $v' = v_1 v v_2$ for some v_1 and v_2.

Definition C.4 ($\mathcal{A}_C$). Given a set C of constraints over s and a, we define $\mathcal{A}_C = (Q, \Delta, C)$ where

- Q is the set of all words $a_1 v a_2$ such that
  - $v \in \mathrm{Prefs}(C)$, and
  - $a_1, a_2 \in \Gamma \cup \{\varepsilon, \hat{}, \$\}$.
- $\Delta \subseteq Q \times (\Gamma \cup \{\hat{}, \$\}) \times Q$ is the set of transitions $v \xrightarrow{a} v'$ where v' is the longest suffix of va such that $v' \in Q$.

Observe that the size of the automaton $\mathcal{A}_C$ is polynomial in the size of C.

A *run* of $\mathcal{A}_C$ over a word with beginning and end marked $a_1 \ldots a_n \in \hat{}\,\Gamma^* \$$ is

$$(v_0, S_0, V_0) \xrightarrow{a_1} (v_1, S_1, V_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (v_n, S_n, V_n)$$

where $v_0 = \varepsilon$ and for all $1 \leq i \leq n$ we have $v_{i-1} \xrightarrow{a_i} v_i$ and $S_i, V_i \subseteq C$ track the satisfied and violated constraints respectively. That is $S_0 = V_0 = \emptyset$, and for all $1 \leq i \leq n$ we have (noting $\hat{}v \preceq v_i$ implies $\hat{}v$ is a prefix of v_i, and similar for $v\$$) $S_i =$

$$\begin{array}{l}
S_{i-1} \cup \{\, \texttt{[} s \,|\, a = v \texttt{]} \in C \mid \hat{}v\$ = v_i \} \ \cup \\
\{\, \texttt{[} s \,|\, a \ \texttt{\~{}=}\ v \texttt{]} \in C \mid \exists a_1 \in \{\hat{}, \textvisiblespace\}, a_2 \in \{\textvisiblespace, \$\} \ .\ a_1 v a_2 \preceq v_i \} \ \cup \\
\{\, \texttt{[} s \,|\, a \ \texttt{|=}\ v \texttt{]} \in C \mid \exists a_2 \in \{\$, \texttt{-}\} \ .\ \hat{}v a_2 \preceq v_i \} \ \cup \\
\{\, \texttt{[} s \,|\, a \ \texttt{\^{}=}\ v \texttt{]} \in C \mid \hat{}v \preceq v_i \} \cup \{\, \texttt{[} s \,|\, a \ \texttt{\$=}\ v \texttt{]} \in C \mid v\$ \preceq v_i \} \ \cup \\
\{\, \texttt{[} s \,|\, a \ \texttt{*=}\ v \texttt{]} \in C \mid v \preceq v_i \}
\end{array}$$

and $V_i =$

$$\begin{array}{l} V_{i-1} \cup \{\texttt{:not([s|a=v])} \in C \mid \hat{}v\$ = v_i\} \cup \\ \left\{\texttt{:not([s|a\textasciitilde=v])} \in C \;\middle|\; \exists \begin{array}{c} a_1 \in \{\hat{}, \sqcup\}, \\ a_2 \in \{\sqcup, \$\} \end{array} . a_1 v a_2 \preceq v_i\right\} \cup \\ \{\texttt{:not([s|a|=v])} \in C \mid \exists a_2 \in \{\$, \texttt{-}\} . \hat{}v a_2 \preceq v_i\} \cup \\ \{\texttt{:not([s|a\hat{}=v])} \in C \mid \hat{}v \preceq v_i\} \cup \\ \{\texttt{:not([s|a\$=v])} \in C \mid v\$ \preceq v_i\} \cup \\ \{\texttt{:not([s|a}\star\texttt{=v])} \in C \mid v \preceq v_i\} . \end{array}$$

Such a run is *accepting* if $S_n = \{[s \,|\, a \text{ op } v] \mid [s \,|\, a \text{ op } v] \in C\}$ and $V_n = \emptyset$. That is, all positive constraints have been satisfied and no negative constraints have been violated.

Short Solutions. We show the existence of short solutions via the following lemma. The proof of this lemma is a simple pumping argument which appears below. Intuitively, if a satisfying word is shorter than $N_s \cdot N_c$ we do not change it. If it is longer than $N_s \cdot N_c$ any accepting run of the automaton on this word must contain a repeated (v, S, V). We can thus pump down this word to ensure that it is shorter than $N_s \cdot N_c$. Then, to ensure it is unique, we pump it up to some unique length of at most $n \cdot N_s \cdot N_c$.

LEMMA C.5 (SHORT ATTRIBUTE VALUES). *Given a sequence of sets of constraint automata $\mathcal{A}_{C_1}, \ldots, \mathcal{A}_{C_n}$ each with at most N_s states and at most N_c constraints in each C_i, if there is a sequence of pairwise unique words $v_1, \ldots, v_n$ such that for all $1 \leq i \leq n$ there is an accepting run of $\mathcal{A}_{C_i}$ over v_i, then there exists such a sequence where the length of each v_i is at most $n \cdot N_s \cdot N_c$.*

To obtain Lemma C.3 (Bounded Attribute Values) we observe that for any subsequence $C_{i_1}, \ldots, Ci_m$ we have $m \cdot N'_s \cdot N'_c \leq n \cdot N_s \cdot N_c$ since $m \leq n$ and the max number of states N'_s and constraints N'_c in the subsequence have $N'_s \leq N_s$ and $N'_c \leq N_c$.

We give the proof of Lemma C.5. That is, given a sequence of sets of constraint automata $\mathcal{A}_{C_1}, \ldots, \mathcal{A}_{C_n}$ each with at most N_s states and at most N_c constraints in each C_i, if there is a sequence of pairwise unique words $v_1, \ldots, v_n$ such that for all $1 \leq i \leq n$ there is an accepting run of $\mathcal{A}_{C_i}$ over v_i, then there exists such a sequence where the length of each v_i is at most $n \cdot N_s \cdot N_c$.

To prove the lemma, take a sequence $v_1, \ldots, v_n$ such that each v_i is unique and accepted by $\mathcal{A}_{C_i}$. We proceed by induction, constructing $v'_1, \ldots, v'_i$ such that each v'_j is unique, accepted by $\mathcal{A}_{C_j}$, and of length ℓ such that either

- $\ell \leq N_s \cdot N_c$ and $v'_j = v_j$, or
- $i \cdot N_s \cdot N_c \leq \ell \leq (i+1) \cdot N_s \cdot N_c$.

When $i = 0$ the result is vacuous. For the induction there are two cases.

When the length ℓ of v_i is such that $\ell \leq N_s \cdot N_c$ we set $v'_i = v_i$. We know v'_i is unique amongst $v'_1, \ldots, v'_i$ since for all $j < i$ either v'_j is longer than v'_i or is equal to v_j and thus distinct from v_i.

When $\ell > N_s \cdot N_c$ we use a pumping argument to pick some v'_i of length ℓ' such that $i \cdot N_s \cdot N_c \leq \ell' \leq (i+1) \cdot N_s \cdot N_c$. This ensures that v'_i is unique since it is the only word whose length lies within the bound. We take the accepting run

$$(u_0, S_0, V_0) \xrightarrow{a_1} (u_1, S_1, V_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (u_\ell, S_n, V_\ell)$$

of v_i and observe that the values of S_j and V_j are increasing by definition. That is $S_j \subseteq S_{j+1}$ and $V_j \subseteq V_{j+1}$. By a standard down pumping argument, we can construct a short accepting run containing only distinct configurations of length bound by $N_s \cdot N_c$. We construct this run by removing all cycles from the original run. This maintains the acceptance condition. Next we obtain an accepted word of length $i \cdot N_s \cdot N_c \leq \ell' \leq (i+1) \cdot N_s \cdot N_c$. Since $\ell > N_s \cdot N_c$ we know there exists at least one configuration (u, S, V) in the short run that appeared twice in the original run. Thus there is a run of the automaton from (u, S, V) back to (u, S, V) which can be bounded by $N_s \cdot N_c$ by the same downward pumping argument as before. Thus, we insert this run into the short run the required number of times to obtain an accepted word v'_i of the required length.

Thus, by induction, we are able to obtain the required short words $v'_1, \ldots, v'_n$ as needed.

C.3.3 Missing definitions for $\mathrm{AttsPres}(\theta, \vec{x})$.

$$
\begin{array}{rcl}
\mathrm{AttsPres}(\,\texttt{[}s\,\texttt{|}\,a\ \texttt{\~{}=}\ v\texttt{]}\,, \vec{x}) & = & \left(\begin{array}{c}
\bigwedge\limits_{1 \le j \le m} \left(\begin{array}{c} x^{s:a}_{i,j} = a_j \wedge \\ \left(\begin{array}{c} x^{s:a}_{i,m+1} = 0 \\ \vee \\ x^{s:a}_{i,m+1} = \sqcup \end{array} \right) \end{array} \right) \\
\vee \\
\bigvee\limits_{1 \le j \le N-m-1} \left(\begin{array}{c} x^{s:a}_{i,j-1} = \sqcup \wedge \\ \bigwedge\limits_{1 \le j' \le m} x^{s:a}_{i,j+j'} = a_j \wedge \\ \left(x^{s:a}_{i,j+m+1} = 0 \vee x^{s:a}_{i,j+m+1} = \sqcup \right) \end{array} \right)
\end{array} \right) \\
\mathrm{AttsPres}(\,\texttt{[}s\,\texttt{|}\,a\ \texttt{|=}\ v\texttt{]}\,, \vec{x}) & = & \bigwedge\limits_{1 \le j \le m} x^{s:a}_{i,j} = a_j \wedge \left(x^{s:a}_{i,m+1} = 0 \vee x^{s:a}_{i,m+1} = \texttt{-} \right) \\
\mathrm{AttsPres}(\,\texttt{[}s\,\texttt{|}\,a\ \texttt{\$=}\ v\texttt{]}\,, \vec{x}) & = & \bigvee\limits_{0 \le j \le N-m-1} \left(\begin{array}{c} \bigwedge\limits_{1 \le j' \le m} x^{s:a}_{i,j+j'} = a_j \wedge \\ x^{s:a}_{i,j+m+1} = 0 \end{array} \right)
\end{array}
$$

C.3.4 Negating Positional Formulas. We need to negate selectors like `:nth-child(`α`n + `β`)`, For completeness, we give the definition of the negation below.

We decompose β according to the period α. I.e. $\beta = \alpha\beta_1 + \beta_2$, where β_1 and β_2 are the unique integers such that $|\beta_2| < |\alpha|$ and $\beta_1\alpha < 0$ implies $\beta_2 \le 0$ and $\beta_1\alpha > 0$ implies $\beta_2 \ge 0$.

Definition C.6 ($\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$*).* Given constants α, β, β_1, and β_2 as above, we define the formula $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$ to be

$$
\begin{array}{c}
(0 \ge \alpha \wedge \overline{x} < \beta) \vee (0 \le \alpha \wedge \overline{x} > \beta) \ \vee \\
\left(\exists \overline{n}\ .\ \exists \overline{\beta}'_2 . \left(\begin{array}{c}
\left|\overline{\beta}'_2\right| < |\alpha| \ \wedge \\
\left(\beta_1\alpha < 0 \Rightarrow \overline{\beta}'_2 \le 0\right) \ \wedge \\
\left(\beta_1\alpha > 0 \Rightarrow \overline{\beta}'_2 \ge 0\right) \ \wedge \\
\overline{\beta}'_2 \ne \beta_2 \wedge \\
\overline{x} = \alpha\overline{n} + \alpha\beta_1 + \overline{\beta}'_2
\end{array} \right) \right)
\end{array}
$$

In the following, whenever we negate a formula of the form $\neg\left(\exists \overline{n}. \overline{x} = \alpha\overline{n} + \beta\right)$ we will use $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$. One can verify that the resulting formula is existential Presburger. We show that our negation of periodic constraints is correct.

PROPOSITION C.7 (CORRECTNESS OF $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$). *Given constants α and β, we have*

$$\neg\left(\exists \overline{n}. \overline{x} = \alpha\overline{n} + \beta\right) \Leftrightarrow \mathrm{NoMatch}(\overline{x}, \alpha, \beta)\ .$$

PROOF. We first consider $\alpha = 0$. Since there is no β'_2 with $\left|\beta'_2\right| < 0$ we have to prove

$$\neg\left(\overline{x} = \beta\right) \Leftrightarrow \left(\overline{x} < \beta\right) \vee \left(\overline{x} > \beta\right)$$

which is immediate.

In all other cases, the conditions on β_2 and β'_2 ensure that we always have $0 < \left|\beta_2 - \beta'_2\right| < |\alpha|$.

If $\exists \overline{n}.\overline{x} = \alpha\overline{n} + \alpha\beta_1 + \beta_2$ then if $\alpha > 0$ it is easy to verify that we don't have $\overline{x} < \beta$ (since $\overline{n} = 0$ gives $\overline{x} = \beta$ as the smallest value of $\overline{x}$) and similarly when $\alpha < 0$ we don't have $\overline{x} < \beta$. To disprove the final disjunct of of $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$ we observe there can be no $\overline{n}'$ s.t. $\overline{x} = \alpha\overline{n}' + \alpha\beta_1 + \beta_2'$ since $\overline{x} = \alpha\overline{n} + \alpha\beta_1 + \beta_2$ and $0 < \left|\beta_2 - \beta_2'\right| < |\alpha|$.

In the other direction, we have three cases depending on the satisfied disjunct of $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$. Consider $\alpha > 0$ and $\overline{x} < \beta$. In this case there is no $\overline{n}$ such that $\overline{x} = \alpha\overline{n} + \alpha\beta_1 + \beta_2 = \alpha\overline{n} + \beta$ since $\overline{n} = 0$ gives the smallest value of $\overline{x}$, which is β. The case is similar for the second disjunct with $\alpha < 0$.

The final disjunct gives some $\overline{n}'$ such that $\overline{x} = \alpha\overline{n}' + \alpha\beta_1 + \beta_2'$ with $0 < \left|\beta_2 - \beta_2'\right| < |\alpha|$. Hence, there can be no $\overline{n}$ with $\overline{x} = \alpha\overline{n} + \alpha\beta_1 + \beta_2$. □

C.3.5 Correctness of Presburger Encoding. We prove soundness and completeness of the Presburger encoding of CSS automata non-emptiness in the two lemmas below.

LEMMA C.8. *For a CSS automaton $\mathcal{A}$, we have*

$$\mathcal{L}(\mathcal{A}) \neq \emptyset \Rightarrow \theta_{\mathcal{A}} \text{ is satisfiable.}$$

PROOF. We take a run of $\mathcal{A}$ and construct a satisfying assignment to the variables in $\theta_{\mathcal{A}}$. That is take a document tree $T = (D, \lambda)$, node $\eta \in D$, and sequence

$$q_0, \eta_0, q_1, \eta_1, \ldots, q_\ell, \eta_\ell, q_{\ell+1} \in (Q \times D)^* \times \left\{q_f\right\}$$

that is an accepting run. We know from Proposition 6.4 (Bounded Types) that T can only use namespaces from $\downarrow(\mathrm{NS})$ and elements from $\downarrow(E)$. Let $t_0, \ldots, t_\ell$ be the sequence of transitions used in the accepting run. We assume (w.l.o.g.) that no transition is used twice. We construct a satisfying assignment to the variables as follows.

- $\overline{q}_i = q_i$ for all $i \leq \ell + 1$ and $\overline{q}_i = q_f$ for all $i > \ell + 1$.
- $\overline{s}_i = \lambda_{\mathrm{S}}(\eta_i)$ for all $i \leq \ell + 1$ ($\overline{s}_i$ can take any value for other values of i).
- $\overline{e}_i = \lambda_{\mathrm{E}}(\eta_i)$ for all $i \leq \ell + 1$ ($\overline{e}_i$ can take any value for other values of i).
- $\overline{p}_i = (p \in \lambda_{\mathrm{P}}(\eta_i))$, for each pseudo-class $p \in P \setminus \{\texttt{:root}\}$ and $i \leq \ell + 1$ (these variables can take any value for $i > \ell + 1$).
- $\overline{n}_i = \iota$, when $\eta_i = \eta'\iota$ for some η' and ι and $1 \leq i \leq \ell + 1$, otherwise $\overline{n}_i$ can take any value.
- $\overline{n}_i^{s:e} = j$, where j is the number of nodes of type $s{:}e$ preceding η_i in the sibling order. That is $\eta_i = \eta'\iota$ for some η' and ι and
$$j = \left|\left\{\eta'\iota' \;\middle|\; \begin{array}{c} \iota' < \iota \;\wedge \eta'\iota' \in D \;\wedge \\ \lambda_{\mathrm{S}}(\eta'\iota') = \lambda_{\mathrm{S}}(\eta) \;\wedge \\ \lambda_{\mathrm{E}}(\eta'\iota') = \lambda_{\mathrm{E}}(\eta) \end{array}\right\}\right| .$$
When $i = 0$ or $i > \ell + 1$ we can assign any value to $\overline{n}_i^{s:e}$.
- $\overline{N}_i = N - \iota$ where $i \leq \ell + 1$ and $\eta = \eta'\iota$ for some η' and ι and N is the smallest number such that $\eta' N \notin D$. For $i = 0$ or $i > \ell + 1$ the variable $\overline{N}_i$ can take any value.
- $\overline{N}_i^{s:e} = j$, where j is the number of nodes of type $s{:}e$ suceeding η_i in the sibling order. That is $\eta_i = \eta'\iota$ for some η' and ι and
$$j = \left|\left\{\eta'\iota' \;\middle|\; \begin{array}{c} \iota' > \iota \;\wedge \eta'\iota' \in D \;\wedge \\ \lambda_{\mathrm{S}}(\eta'\iota') = \lambda_{\mathrm{S}}(\eta) \;\wedge \\ \lambda_{\mathrm{E}}(\eta'\iota') = \lambda_{\mathrm{E}}(\eta) \end{array}\right\}\right| .$$
When $i = 0$ or $i > \ell + 1$ we can assign any value to $\overline{N}_i^{s:e}$.
- Assignments to $x_{i,j}^{s:a}$ are discussed below.

It remains to prove that the given assignment satisfies the formula.

Recall

$$\theta_{\mathcal{A}} = \left(\begin{array}{c} \overline{q}_0 = q^{\text{in}} \wedge \overline{q}_n = q_f \wedge \\ \bigwedge\limits_{0 \leq i < n} \left(\text{Tran}(i) \vee \overline{q}_i = q_f \right) \wedge \\ \text{Consistent} \end{array} \right).$$

The first two conjuncts follow immediately from our assignment to $\overline{q}_i$ and that the chosen run was accepting. Next we look at the third conjunct and simultaneously prove Consistent_n. When $i \geq \ell + 1$ we assigned q_f to $\overline{q}_i$ and can choose any assignment that satisfies Consistent_n. Otherwise we show we satisfy $\text{Tran}(i)$ by showing we satisfy $\text{Tran}(i, t_i)$. We also show Consistent_n is satsfied by induction, noting it is immediate for $i = 0$ and that for $i = 1$ we must have either the first orlast case which do not depend on the induction hypothesis. Consider the form of t_i.

(1) When $t_i = q_i \xrightarrow[\sigma]{\downarrow} q_{i+1}$ we immediately confirm the values of $\overline{q}_i$, $\overline{q}_{i+1}$, $\overline{n}_{i+1}$, $\overline{n}^{s:e}_{i+1}$ satisfy the constraint. Similarly for $\overline{\neg\texttt{:empty}}_i$ since we know $\texttt{:empty} \notin \lambda_{\mathrm{P}}(\eta_i)$. We defer the argument for $\text{Pres}(\sigma, i)$ until after the case split. That Consistent_n is satisfied can also be seen directly.

(2) When $t_i = q_i \xrightarrow[\sigma]{\rightarrow} q_{i+1}$ we know $\eta_i = \eta'\iota$ and $\eta_{i+1} = \eta'(\iota + 1)$ for some η' and ι. We can easily check the values of $\overline{q}_i$, $\overline{q}_{i+1}$, $\overline{n}_{i+1}$, $\overline{N}_i$, $\overline{n}^{s:e}_{i+1}$, and $\overline{N}^{s:e}_{i+1}$ satisfy the constraint. We defer the argument for $\text{Pres}(\sigma, i)$ until after the case split. To show Consistent_n we observe $\overline{n}_{i+1}$ is increased by 1 and only one $\overline{n}^{s:e}_{i+1}$ is increased by 1, the others being increased by 0. Similarly for $\overline{N}_i$ and $\overline{n}^{s:e}_{i+1}$. Hence the result follows from induction.

(3) When $t_i = q_i \xrightarrow[*]{\rightarrow_+} q_{i+1}$ we know $\eta_i = \eta'\iota$ and $\eta_{i+1} = \eta'(\iota')$ for some η', ι, and $\iota < \iota'$. We can easily check the values of $\overline{q}_i$, $\overline{q}_{i+1}$, $\overline{n}_{i+1}$, $\overline{N}_i$, $\overline{n}^{s:e}_{i+1}$, and $\overline{N}^{s:e}_{i+1}$ satisfy the constraint. We defer the argument for $\text{Pres}(\sigma, i)$ until after the case split. To satisfy the constraints over the position variables, we observe that values for $\overline{\delta}$ and $\overline{\delta}_{s:e}$ can be chosen easily for the specified assignment. Combined with induction this shows Consistent_n as required.

(4) When $t_i = q_i \xrightarrow[\sigma]{\circ} q_{i+1}$

We can easily check the values of $\overline{q}_i$ and $\overline{q}_{i+1}$. We defer the argument for $\text{Pres}(\sigma, i)$ until after the case split. By induction we immediately obtain Consistent_n.

We show $\text{Pres}(\sigma, i)$ is satisfied for each η_i and σ labelling t_i. Take a node η and $\sigma = \tau\Theta$ from this sequence. Note η satisfies σ since thw run is accepting. Recall

$$\text{Pres}(\tau\Theta, i) = \left(\begin{array}{c} \text{Pres}(\tau, i) \wedge \\ \left(\bigwedge\limits_{\theta \in \text{NoAtts}(\Theta)} \text{Pres}(\theta, i) \right) \wedge \\ \text{AttsPres}(\tau\Theta, i) \end{array} \right).$$

From the type information of η we immediately satisfy $\text{Pres}(\tau, i)$.

For a positive $\theta \in \Theta$ there are several cases. If $\theta = \texttt{:root}$ then we know we are in η_0 and the encoding is $\top$. If θ is some other pseudo class p then the encoding of θ is $\overline{p}_i$ and we assigned true to this variable. For $\texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)}$ and $\texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)}$ satisfaction of the encoding follows immediately from η satisfying θ and our assignment to $\overline{n}_i$ and $\overline{N}_i$. We satisfy the encodings of $\texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)}$, $\texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)}$, `:only-child`, and `:only-of-type` similarly. The latter follow since an only child is position

1 from the start and end, and an only of type node has 0 strict predecessors or successors of the same type.

For a negative $\theta \in \Theta$ there are several cases. If $\theta = $ `:not(:root)` then we know we are not in η_0 and the encoding is $\top$. If θ is the negation of some other pseudo class p then the encoding of θ is $\neg\overline{p}_i$ and we assigned false to this variable. For the selectors `:not(:nth-child(`α`n + `β`))` and the opposite selector `:not(:nth-last-child(`α`n + `β`))` satisfaction of the encoding follows immediately from η satisfying θ, our assignment to $\overline{n}_i$ and $\overline{N}_i$ as well as Proposition C.7 (Correctness of $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$). We satisfy encodings of `:not(:nth-of-type(`α`n + `β`))` and of `:not(:nth-last-of-type(`α`n + `β`))` in a likewise fashion. For the remaining cases of `:not(:only-child)`, and `:not(:only-of-type)` the property follows since, for the former the node must either not be position 1 from the start or end, and for the latter a not only of type node has more than 0 strict predecessors or successors of the same type.

Next, to satisfy $\mathrm{AttsPres}(\tau\Theta, i)$ we have to satisfy a number of conjuncts. First, if we have a word $a_1 \ldots a_n$ we assign it to the variables $x^{s:a}_{i,j}$ (where η is the ith in the run and j ranges over all word positions within the cimputed bound) ny assigning $x^{s:a}_{i,j} = a_j$ when $j \leq n$ and $x^{s:a}_{i,j} = 0$ otherwise.

In all cases below, it is straightforward to observe that it a word (within the computed length bound) satisfies `[`s`|`a op v`]` or `:not([s|a op v])` then the encoding $\mathrm{AttsPres}_{s:a}(\texttt{[}s\,|\,a \text{ op } v\texttt{]}, i)$ or $\neg\mathrm{AttsPres}_{s:a}(\texttt{[}s\,|\,a \text{ op } v\texttt{]}, i)$ is satisfied by our variable assignment. Similarly $\mathrm{Nulls}(\vec{x})$ is straightforwardly satisfied. Hence, if a word satisfies C then our assignment to the variables means $\mathrm{AttsPres}_{s:a}(C, i)$ is also satisfied.

There are a number of cases of conjuncts for attribute selectors. The simplest is for sets Θ^s_a where we see immediately that all constraints are satisfied for $\lambda_{\mathbb{A}}(\eta)(s, a)$ and hence we assign this value to the appropriate variables and the conjuct is satisfied also. For each `[`a`]` and `[`a op v`]` $\in \Theta$ we have in the document some namespace s such that $\lambda_{\mathbb{A}}(\eta)(s, a)$ satisfies the attribute selector and all negative selectors applying to all namespaces. Let s' be the fresh name space assigned to the selector during the encoding and C be the full set of constraints belonging to the conjunct (i.e. including negative ones). We assign to the variable $x^{s:a}_{i,j}$ the jth character of $\lambda_{\mathbb{A}}(\eta)(s, a)$ (where η is the ith in the run) and satisfy the conjuct as above. Note here that a single value of $s{:}a$ is assigned to several $s'{:}a$. This is benign with respect to the global uniqueness required by ID attributes because each copy has a different namespace.

Finally, we have to satisfy the consistency constraints. We showed $\mathrm{Consistent}_n$ above. The remaining consistency constraints are easily seen to be satisfied: $\mathrm{Consistent}_i$ because each ID is unique causing at least one pair of characters to differ in every value; $\mathrm{Consistent}_p$ since it encodes basic consistency constraints on the appearence of pseudo elements in the tree.

Thus, we have satisfied the encoded formula, completing the first direction of the proof. □

LEMMA C.9. *For a CSS automaton $\mathcal{A}$, we have*

$$\theta_{\mathcal{A}} \text{ is satisfiable.} \Rightarrow \mathcal{L}(\mathcal{A}) \neq \emptyset$$

PROOF. Take a satisfying assignment ρ to the free variables of $\theta_{\mathcal{A}}$. We construct a tree and node (T, η) as well as a run of $\mathcal{A}$ accepting (T, η).

We begin by taking the sequence of states $q_0, \ldots, q_{\ell+1}$ which is the prefix of the assignment to $\overline{q}_0, \ldots, \overline{q}_n$ where q_ℓ is the first occurrence of q_f. We will construct a series of transitions $t_0, \ldots, t_\ell$ with $t_i = q_i \xrightarrow[\sigma_i]{d_i} q_{i+1}$ for all $0 \leq i \leq \ell$. We will define each d_i and σ_i, as well as construct T and η by induction. We construct the tree inductively, then show σ_i is satisfied for each i.

At first let T_0 contain only a root node. Thus η_0 is necessarily this root. Throughout the proof we label each η_i as follows.

- $\lambda_{\mathrm{S}}(\eta_i) = \rho(\overline{s}_i)$ (i.e. we assign the value given to $\overline{s}_i$ in the satisfying assignment).
- $\lambda_{\mathrm{E}}(\eta_i) = \rho(\overline{e}_i)$.
- $\lambda_{\mathrm{P}}(\eta_i) = \left(\begin{array}{c} \{p \mid p \in P \setminus \{\texttt{:root}\} \wedge \rho(\overline{p}_i) = \top\} \cup \\ \{\texttt{:root} \mid i = 0\} \end{array} \right)$.
- $\lambda_{\mathrm{A}}(\eta_i)(s, a) = \rho\left(x^{s:a}_{i,1} \dots x^{s:a}_{i,N}\right)$ where $\rho\left(x^{s:a}_{i,1} \dots x^{s:a}_{i,N}\right)$ is the word obtained by stripping all of the null characters from $\rho\left(x^{s:a}_{i,1}\right) \dots \rho\left(x^{s:a}_{i,N}\right)$.

We pick t_i as the transition corresponding to a satisfied disjunct of $\mathrm{Tran}(i)$ (of which there is at least one since $q_i \neq q_f$ when $i \leq \ell$). Thus, take $t_i = q_i \xrightarrow[\sigma_i]{d_i} q_i$. We proceed by a case split on d_i. Note only cases $d_i =\downarrow$ and $d_i = \circ$ may apply when $i = 0$.

- When $d_i =\downarrow$ we build T_{i+1} as follows. First we add the leaf node $\eta_{i+1} = \eta_i 1$. Then, if $i > 0$, we add siblings appearing after η_i with types required by the last of type information. That is, we add $\rho\left(\overline{N}_i\right) - 1 = \sum\limits_{s:e \in E} \rho\left(\overline{N}^{s:e}_i\right)$ siblings appearing after η_i. In particular, for each s and e we add $\rho\left(\overline{N}^{s:e}_i\right)$ new nodes. Letting $\eta_i = \eta\iota$ each of these new nodes η' will have the form $\eta\iota'$ with $\iota' > \iota$. We set $\lambda_{\mathrm{S}}(\eta') = s$, $\lambda_{\mathrm{E}}(\eta') = e$, $\lambda_{\mathrm{P}}(\eta') = \emptyset$, $\lambda_{\mathrm{A}}(\eta') = \emptyset$.
- When $d_i =\rightarrow$ we build T_{i+1} by adding a single node to T_i. When $\eta_i = \eta\iota$ we add $\eta_{i+1} = \eta\,(\iota + 1)$ with the labelling as above.
- When $d_i =\rightarrow_+$ we build T_{i+1} as follows. We add $\rho\left(\overline{\delta}\right) = (\rho(\overline{n}_{i+1}) - \rho(\overline{n}_i))$ new nodes of the form $\eta\iota'$ where $\eta_i = \eta\iota$ and $\rho(\overline{n}_i) = \iota < \iota' \leq \rho(\overline{n}_i)$. Let η_{i+1} be $\eta\rho(\overline{n}_i)$ labelled as above. For the remaining new nodes, for each s and e, we label $\rho\left(\overline{\delta}_{s:e}\right)$ of the new nodes η' with $\lambda_{\mathrm{S}}(\eta') = s$, $\lambda_{\mathrm{E}}(\eta') = e$, $\lambda_{\mathrm{P}}(\eta') = \emptyset$, $\lambda_{\mathrm{A}}(\eta') = \emptyset$. Note $\mathrm{Consistent}_n$ ensures we have enough new nodes to partition like this.
- When $d_i = \circ$ and $i = 0$ we have completed building the tree. If $i > 0$, we add siblings appearing after η_i with types required by the last of type information exactly as in the case of $d =\downarrow$ above.

The tree and node we require are the tree and node obtained after reaching some $d_i = \circ$, for which we necessairily have $i = \ell$ since $\circ$ must be and can only be used to reach q_f. In constructing this tree we have almost demonstrated an accepting run of $\mathcal{A}$. To complete the proof we need to argue that all σ_i are satisfied by η_i and that the obtained is valid. Let $\tau\Theta = \sigma_i$.

To check τ we observe that $\mathrm{Pres}(\tau, i)$ constrains $\overline{s}_i$ and $\overline{e}_i$ to values, which when assigned to η_i as above mean η_i directly satisfies τ.

Now, take some $\theta \in \Theta$. In each case we argue that $\mathrm{Pres}(\tau, i)$ ensures the needed properties. Note this is straightforward for the attribute selectors due to the directness of the Presburger encoding. Consider the remaining selectors.

First assume θ is positive. If it is `:root` then we must have $i = 0$ and η_i is the root node as required. For other pseudo classes p we asserted $\overline{p}_i$ hence we have $p \in \lambda_{\mathrm{P}}(\eta_i)$. The encoding of the remaining positive constraints can only be satisfied when $i > 0$. That is, η_i is not the root node.

For `:nth-child(`α`n + `β`)` observe we constructed T such that $\eta_i = \eta\rho(\overline{n}_i)$ for some η. From the defined encoding of $\mathrm{Pres}($`:nth-child(`α`n + `β`)`$, i)$ we directly obtain that η_i satisfies `:nth-child(`α`n + `β`)`. Similarly for `:nth-last-child(`α`n + `β`)` as we always pad the end of the sibling order to ensure the correct number of succeeding siblings.

For `:nth-of-type(`α`n + `β`)` and `:nth-last-of-type(`α`n + `β`)` selectors, by similar arguments to the previous selectors, we have ensured that there are enough preceeding or succeeding nodes (along with the directness of their Presburger encoding) to ensure these selectors are satisfied by η_i in T.

For `:only-child` we know there are no other children since $\rho(\overline{n}_i) = \rho\left(\overline{N}_i\right) = 1$. Finally for the selector `:only-of-type` we know there are no other children of the same type since $\rho(\overline{n}_i^{s:e}) = \rho\left(\overline{N}_i^{s:e}\right) = 0$ where η_i has type $s{:}e$.

When θ is negative there are several cases. If it is `:not(:root)` then we must have $i > 0$ and η_i is not the root node. For other pseudo classes p we asserted $\neg\overline{p}_i$ hence we have $p \notin \lambda_{\mathrm{P}}(\eta_i)$. The encoding of the remaining positive constraints are always satisfied on the root node. That is, $i = 0$. When η_i is not the root node we have $i > 0$.

For `:not(:nth-child(`α`n + `β`))` observe we constructed T such that $\eta_i = \eta\rho(\overline{n}_i)$ for some η. From the definition of $\mathrm{Pres}($`:not(:nth-child(`α`n + `β`))`$, i)$ we obtain that η_i does not satisfy the required selector `:nth-child(`α`n + `β`)` via Proposition C.7 (Correctness of $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$). Similarly for the last child selector

`:not(:nth-last-child(`α`n + `β`))` .

For `:not(:nth-of-type(`α`n + `β`))` and `:not(:nth-last-of-type(`α`n + `β`))`, by similar arguments to the previous selectors, we have ensured that there are enough preceeding or succeeding nodes (along with their Presburger encodings and Proposition C.7 (Correctness of $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$)) to ensure these selectors are satisfied by η_i in T.

For `:not(:only-child)` we know there are some other children since $\rho(\overline{n}_i) > 1$ or $\rho\left(\overline{N}_i\right) > 1$. Finally for `:not(:only-of-type)` we know there are other children of the same type since $\rho(\overline{n}_i^{s:e}) > 0$ or $\rho\left(\overline{N}_i^{s:e}\right) > 0$ where η_i has type $s{:}e$.

Thus we have an accepting run of $\mathcal{A}$ over some (T, η). However, we finally have to argue that T is a valid document tree. This is enforced by $\mathrm{Consistent}_i$ and $\mathrm{Consistent}_p$.

First, $\mathrm{Consistent}_i$ ensures all IDs satisfying the Presburger encoding are unique. Since we transferred these values directly to T our tree also has unique IDs.

Next, we have to ensure properties such as no node is both active and inactive. These are all directly taken care of by $\mathrm{Consistent}_p$. Thus, we are done. □

D ADDITIONAL MATERIAL FOR THE EXPERIMENTS SECTION

D.1 Optimised CSS Automata Emptiness Check

The reduction presented in Section 5 proves membership in NP. However, the formula constructed is quite large even for the intersection of two relatively small selectors. Moreover, selectors generally do no assert complex properties, so for most transitions, the full power of existential Presburger arithmetic is not needed. Hence, only a small part of each formula requires complex reasoning, while the remainder of the problem is better and easily solved with direct knowledge of the automata.

In this section we present an alternative algorithm. In essence it is a backwards reachability algorithm for deciding non-emptiness of a CSS automaton. Instead of constructing a single large query that requires a non-trivial solve time, the backwards reachability algorithm only makes small queries to the SAT solver to enforce constraints that are not simply enforced by a standard automaton algorithm.

The idea is that the automaton collects constraints on the node positions required to satisfy the nth-child (sibling) constraints as it performs its backwards search. It also tracks extra information to ensure it does not get stuck in a loop, at most one node is labelled `:target`, and all `ids` are unique. Each time the automaton takes a transition labelled $\downarrow$ it checks whether the current set of sibling constraints is satisfiable. If so, the automaton can move up to the parent node and begin with a fresh set of sibling constraints. If not, the automaton cannot execute the transition. Once the initial state has been reached, it just remains to check whether the `id` constraints are satisfiable. If they are, a witness to non-emptiness has been found.

The algorithm is a worklist algorithm, where the worklist consists of tuples of the form

$$\left(q, b_{\mathrm{root}}, b_{\mathrm{sib}}, b_{\mathrm{targ}}, \mathrm{Ts}, C_{\mathtt{id}}, C_{\mathrm{pos}}, i\right)$$

where

- q is the state reached so far,
- b_{root} is a boolean indicating whether the current node has to be the root,
- b_{sib} is a boolean indicating whether the current node has to have siblings,
- b_{targ} is a boolean indicating whether a node marked `:target` has been seen on the run so far,
- Ts is the set of transitions seen on the current state (recall all loops are self-loops, so cycle detection can be implemented using Ts),
- $C_{\mathtt{id}}$ is the set of constraints on `id` attributes in the run so far,
- C_{pos} is the set of constraints on node positions on the current level of the tree (that is, for the assertion of nth-child constraints),
- i is the index position in the run (akin to the use of indices in the Presburger encoding).

The initial worklist contains a single element

$$\left(q_f, \bot, \bot, \bot, \emptyset, \emptyset, \emptyset, n\right)$$

where n is the number of transitions of the CSS automaton. Note, the final element of the tuple will always range between 1 and n since we decrement this counter whenever we take a new transition, and each transition may only be visited once.

In the following, we partition sets of node selector elements into sets containing pseudo-classes, attribute selectors, and positional selectors. That is, given a node selector $\tau\Theta$ we write

- $\mathrm{Atts}(\Theta)$ for the elements of Θ of the form θ or `:not(`θ`)` where θ is of the form `[`$s \mid a$`]` or `[`$s \mid a \text{ op } v$`]`,
- $\mathrm{Pos}(\Theta)$ for the elements of Θ of the form θ or `:not(`θ`)` where θ is of the form

  `:nth-child(`α`n + `β`)`,
  `:nth-last-child(`α`n + `β`)`,
  `:nth-of-type(`α`n + `β`)`,
  `:nth-last-of-type(`α`n + `β`)`
  , `:only-child`, or
  `:only-of-type`,

- $\mathrm{Pseudo}(\Theta)$ for the elements of Θ of the form θ or `:not(`θ`)` where θ is of the form

  `:link`, `:visited`, `:hover`, `:active`, `:focus`, `:target`,
  `:enabled`, `:disabled`, `:checked`, `:root`, or `:empty` .

If the worklist is empty, we terminate, and return that the automaton is empty.

If it is not empty, we take an arbitrary element

$$\left(q', b_{\mathrm{root}}, b_{\mathrm{sib}}, b_{\mathrm{targ}}, \mathrm{Ts}, C_{\mathtt{id}}, C_{\mathrm{pos}}, i\right)$$

and for each transition

$$t = q \xrightarrow[\sigma]{d} q'$$

with $\sigma = \tau\Theta$ we add to the worklist

$$\left(q, b'_{\mathrm{root}}, b'_{\mathrm{sib}}, b'_{\mathrm{targ}}, \mathrm{Ts}', C'_{\mathtt{id}}, C'_{\mathrm{pos}}, i'\right)$$

where $i' = i-1$ is a fresh index, and when certain conditions are satisfied. We detail these conditions and the definition of the new tuple below. We begin with general conditions and definitions, then describe those specific to the value of d.

In all cases, we can only add a new tuple if

(1) $t \notin \mathrm{Ts}$,
(2) $\mathrm{AttsPres}(\tau\Theta, i')$ is satisfiable,
(3) $\mathrm{Pseudo}(\Theta)$ is satisfiable – that is, we do not have $p \in \Theta$ and `:not(p)` $\in \Theta$ for some pseudo-class p, and, moreover, we do not have
   - `:link` $\in \Theta$ and `:visited` $\in \Theta$, or
   - `:enabled` $\in \Theta$ and `:disabled` $\in \Theta$, or
   - `:root` $\in \Theta$ and $b_{\mathrm{sib}} = \top$.

In all cases, we define

- $b'_{\mathrm{root}} = \begin{cases} \top & \texttt{:root} \in \Theta \\ b_{\mathrm{root}} & d = \circ \\ \bot & \text{otherwise,} \end{cases}$
- $b'_{\mathrm{sib}} = \begin{cases} \top & d \in \{\rightarrow, \rightarrow_{+}\} \\ b_{\mathrm{sib}} & d = \circ \\ \bot & \text{otherwise,} \end{cases}$
- $b'_{\mathrm{targ}} = b_{\mathrm{targ}} \vee (\texttt{:target} \in \Theta)$,
- $\mathrm{Ts}' = \begin{cases} \mathrm{Ts} \cup \{t\} & q = q' \\ \emptyset & \text{otherwise,} \end{cases}$
- $C'_{\mathtt{id}} = C_{\mathtt{id}} \cup C''_{\mathtt{id}}$ where $C''_{\mathtt{id}}$ is the set of all clauses $\mathrm{AttsPres}_{s:\mathtt{id}}(C, i')$ appearing in $\mathrm{AttsPres}(\tau\Theta, i')$ for some s and C.

Next, we give the conditions and definitions dependent on d. To do so we need to define the set of positional constraints derived from $\mathrm{Pos}(\Theta)$. We use a slightly different encoding to the previous section. We use a variable $\overline{n}_i$ encoding that the node is the $\overline{n}_i$th child of the parent, and $\overline{n}_i^{s:a}$ counting the number of nodes of type $s{:}a$ to the left of the current node (exclusive). To encode "last of" constraints, we use the variable $\overline{N}$ to encode the total number of siblings of the current node (inclusive), and $\overline{N}_{s:a}$ to encode the total number of siblings of the given type (inclusive).

That is, when $b'_{\mathrm{root}} = \top$ let

- $C''_{\mathrm{pos}} = \{\bot\}$ if $b'_{\mathrm{sib}} = \top$,
- $C''_{\mathrm{pos}} = \{\bot\}$ if there is some $\theta \in \mathrm{Pos}(\Theta)$ that is not of the form `:not(`θ'`)` for some θ', and
- $C''_{\mathrm{pos}} = \{\top\}$ otherwise,

and when $b'_{\text{root}} = \bot$ let $C''_{\text{pos}} =$

$$\begin{array}{l}
\{\exists \overline{n} \,.\, \overline{n}_{i'} = \alpha\overline{n} + \beta \mid \texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)} \in \Theta\} \cup \\
\left\{\exists \overline{n} \,.\, \overline{N} - \overline{n}_{i'} - 1 = \alpha\overline{n} + \beta \;\middle|\; \texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)} \in \Theta\right\} \cup \\
\left\{ \begin{array}{c} \overline{s}_{i'}\text{:}\overline{e}_{i'} = s\text{:}e \Rightarrow \\ \exists \overline{n} \,.\, \overline{n}^{s:e}_{i'} = \alpha\overline{n} + \beta \end{array} \;\middle|\; \begin{array}{c} s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E) \wedge \\ \texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)} \in \Theta \end{array} \right\} \cup \\
\left\{ \begin{array}{c} \overline{s}_{i'}\text{:}\overline{e}_{i'} = s\text{:}e \Rightarrow \\ \exists \overline{n} \,.\, \overline{N}_{s:e} - \overline{n}^{s:e}_{i'} - 1 = \alpha\overline{n} + \beta \end{array} \;\middle|\; \begin{array}{c} s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E) \wedge \\ \texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)} \in \Theta \end{array} \right\} \cup \\
\{\text{NoMatch}(\overline{n}_{i'}, \alpha, \beta) \mid \texttt{:not(:nth-child(}\alpha\texttt{n + }\beta\texttt{))} \in \Theta\} \cup \\
\left\{\text{NoMatch}\left(\overline{N} - \overline{n}_{i'} - 1, \alpha, \beta\right) \;\middle|\; \texttt{:not(:nth-last-child(}\alpha\texttt{n + }\beta\texttt{))} \in \Theta\right\} \cup \\
\left\{ \begin{array}{c} \overline{s}_{i'}\text{:}\overline{e}_{i'} = s\text{:}e \Rightarrow \\ \text{NoMatch}(\overline{n}^{s:e}_{i'}, \alpha, \beta) \end{array} \;\middle|\; \begin{array}{c} s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E) \wedge \\ \texttt{:not(:nth-of-type(}\alpha\texttt{n + }\beta\texttt{))} \in \Theta \end{array} \right\} \cup \\
\left\{ \begin{array}{c} \overline{s}_{i'}\text{:}\overline{e}_{i'} = s\text{:}e \Rightarrow \\ \text{NoMatch}\left(\overline{N}_{s:e} - \overline{n}^{s:e}_{i'} - 1, \alpha, \beta\right) \end{array} \;\middle|\; \begin{array}{c} s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E) \wedge \\ \texttt{:not(:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{))} \in \Theta \end{array} \right\} \cup \\
\left\{\overline{N} > \overline{n}_{i'}\right\} \cup \left\{ \overline{n}_{i'} = \sum_{\substack{s \in \rfloor(\text{NS}) \\ e \in \rfloor(E)}} \overline{n}^{s:e}_{i'} \right\} \cup \\
\left\{\overline{s}_{i'}\text{:}\overline{e}_{i'} = s\text{:}e \Rightarrow \overline{N}_{s:e} > \overline{n}^{s:e}_{i'} \;\middle|\; s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E)\right\} \cup \\
\left\{\overline{s}_{i'}\text{:}\overline{e}_{i'} \neq s\text{:}e \Rightarrow \overline{N}_{s:e} \geq \overline{n}^{s:e}_{i'} \;\middle|\; s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E)\right\} \, .
\end{array}$$

Then we have the following.

- When $d = \circ$, we set

$$\begin{array}{rcl} C'_{\text{pos}} & = & C_{\text{pos}} \cup C''_{\text{pos}} \cup \\ & & \{\overline{n}_{i'} = \overline{n}_i\} \cup \{\overline{s}_{i'}\text{:}\overline{e}_{i'} = \overline{s}_i\text{:}\overline{e}_i\} \cup \\ & & \{\overline{n}^{s:e}_{i'} = \overline{n}^{s:e}_i \mid s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E)\} \, . \end{array}$$

- When $d = \downarrow$,
  - we require C_{pos} is satisfiable, $\neg b_{\text{root}}$, and `:empty` $\notin \Theta$,
  - we set $C'_{\text{pos}} = C''_{\text{pos}}$.
- When $d = \rightarrow$,
  - we require $\neg b_{\text{root}}$ and `:root` $\notin \Theta$,
  - we set

$$\begin{array}{rcl} C'_{\text{pos}} & = & C_{\text{pos}} \cup C''_{\text{pos}} \cup \{\overline{n}_i = \overline{n}_{i'} + 1\} \cup \\ & & \{\overline{s}_{i'}\text{:}\overline{e}_{i'} = s\text{:}e \Rightarrow \overline{n}^{s:e}_i = \overline{n}^{s:e}_{i'} + 1 \mid s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E)\} \cup \\ & & \{\overline{s}_{i'}\text{:}\overline{e}_{i'} \neq s\text{:}e \Rightarrow \overline{n}^{s:e}_i = \overline{n}^{s:e}_{i'} \mid s \in \rfloor(\text{NS}) \wedge e \in \rfloor(E)\} \, . \end{array}$$

- When $d = \rightarrow_+$
  - we require $\neg b_{\text{root}}$ and `:root` $\notin \Theta$,
  - we set

$$C'_{\text{pos}} = C_{\text{pos}} \cup C''_{\text{pos}} \cup \left\{ \exists \overline{\delta}, \left(\overline{\delta}_{s:e}\right)_{\substack{s \in \rfloor(\text{NS}) \\ e \in \rfloor(E)}} \,.\, \left(\begin{array}{c} \overline{n}_i = \overline{n}_{i'} + \overline{\delta} \wedge \overline{\delta} \geq 1 \wedge \\ \bigwedge_{\substack{s \in \rfloor(\text{NS}) \\ e \in \rfloor(E)}} \left(\begin{array}{c} \overline{n}^{s:e}_i = \overline{n}^{s:e}_i + \overline{\delta}_{s:e} \wedge \\ (\overline{s}_{i'}\text{:}\overline{e}_{i'} = s\text{:}e) \Rightarrow \overline{\delta}_{s:e} \geq 1 \end{array} \right) \wedge \\ \overline{\delta} = \sum_{\substack{s \in \rfloor(\text{NS}) \\ e \in \rfloor(E)}} \overline{\delta}_{s:e} \end{array} \right) \right\} .$$

Moreover, if we were able to add the tuple to the worklist, and we have

- $q = q^{\text{in}}$,
- C'_{pos} is satisfiable, and
- The ID constraints are satisfiable,

then the algorithm terminates, reporting that the automaton is non-empty. To check the ID constraints are satisfiable, we test satisfiability of the following formula. Let N be the bound on the length of ID values, as derived in the previous section. We assert

$$\bigwedge_{\theta \in C'_{\mathtt{id}}} \theta \wedge \bigwedge_{\substack{s \in \mathrm{NS} \\ 1 \leq i_1, i_2 \leq n}} \bigvee_{1 \leq j \leq N} x^{s:\mathtt{id}}_{i_1, j} \neq x^{s:\mathtt{id}}_{i_2, j} .$$

That is, we assert all ID conditions are satisfied, and all IDs are unique.

D.2 Sources of the CSS files used in our experiments

We collected 71 CSS files from 40 global websites for our experiments. These websites cover the 20 most popular sites listed on Alexa (Alexa Internet 2017), which are Google, YouTube, Facebook, Baidu, Wikipedia, Yahoo!, Reddit, Google India, Tencent QQ, Taobao, Amazon, Tmall, Twitter, Google Japan, Sohu, Windows Live, VK, Instagram, Sina, and 360 Safeguard. Note that we excluded Google India and Google Japan from our collection as we found the two sites shared the same CSS files with Google. We further collected CSS files from 12 well-known websites ranked between 21 and 100 on the same list, including LinkedIn, Yahoo! Japan, Netflix, Imgur, eBay, WordPress, MSN, Bing, Tumblr, Microsoft, IMDb, and GitHub. Our examples also contain CSS files from 10 smaller websites, including Arch Linux, arXiv, CNN, DBLP, Google News, Londonist, New York Times, NetworkX, OpenStreetMap, and W3Schools. These examples were used in the testing and development of our tool.