Formalisation and Static Analysis of CSS3 Selectors

Matthew Hague[1] and Anthony Widjaja Lin[2]

[1] Royal Holloway, University of London
[2] Yale-NUS College, Singapore

Abstract

Cascading Style Sheets (CSS) is the de facto language for styling a webpage, as developed and maintained by World Wide Web Consortium (W3C). Together with HTML and JavaScript, CSS has played an integral role in modern web technologies. At the heart of CSS are CSS selectors, which are formal specifications of which nodes in a given document object model (DOM) are to be selected for declarations of certain visual properties (e.g. set colour to red). Despite the apparent simplicity of CSS selectors, CSS selectors allow many complex features, e.g., tree traversals, string constraints on attributes, constraints for counting the number of children, and restricted negations. This paper describes the first precise formalisation of CSS Level 3 (CSS3) selector language, currently the stable version of the language. Unlike previous formalisations, our formalisation attempts to capture CSS3 *in its entirety*, as specified in the latest W3C Recommendation. We then study two fundamental static analysis problems which have found applications in optimisation of CSS files: (1) satisfiability: given a CSS selector, decide whether it can ever select a node in some DOM tree (2) intersection: given two CSS selectors, decide whether they can ever select the same node in some DOM tree. The second problem, in particular, has immediate applications in minimisation of CSS files by refactoring similar styling rules. We show that they are decidable and, in fact, NP-complete. Furthermore, we show that both problems can be reduced in polynomial-time to satisfiability of quantifier-free Presburger formulas, for which highly optimised SMT solvers are available.

1 Introduction

Cascading Style Sheets (CSS), together with HTML and JavaScript, has been a cornerstone language for the web since its inception in 1996. The language facilitates a clear separation between presentation and web content, and plays a pivotal role in the design of a semantically meaningful HTML.

CSS is widely perceived to be a simple language. One simply needs to define a sequence of *(styling) rules*, each containing a set of *(node) selectors* (e.g. `h1, h2`, which select `h1` and `h2` HTML elements in a given document) and a set of *property declarations* (e.g. `font-weight: normal;` and `margin-bottom: 10px;`) that will be assigned to *all* selected nodes[1]. The power of CSS comes from its powerful selectors in that they can express navigational properties over web documents, constraints for counting the number of children, string constraints for reasoning about attributes, conjunctions, and negations, to name a few. For example, the CSS selector

```
.planet:nth-child(7n+1) .moon:not(.water)
```

selects all nodes with class `.moon` but not class `.water` such that it has an ancestor with class `.planet` such that it is the ith child of the parent where $i \equiv 1 \pmod 7$. In this respect, CSS selectors exhibit a striking similarity to XPath [1] which is a popular query language for selecting

[1]For readers unfamiliar with CSS, we provide a primer in Section 2.

nodes or node-sets in an XML document. Both CSS and XPath are defined and maintained by World Wide Web Consortium (W3C).

Despite the apparent simplicity of the language, CSS selectors can be quite subtle. For example, let us take the latest evolution of the CSS language, i.e., CSS3 [7]. Consider the following properties over web documents:

(1) An element `div` with at least one child
(2) An element `div` with no more than three children
(3) An element `div` with at least one child with class `.a`.
(4) An element with class `.a`, and it is the second child, and it does not have a sibling at position 3.
(5) An element `h3` with an ancestor with an ID attribute that matches the regular language $(ab)^*$.
(6) An element `h2` whose ID attribute matches the regular language $(aa)^*$.

It might come as a surprise that (1), (4), (5) are expressible as CSS selectors, while (2), (3), and (6) are not expressible as CSS selectors. [See Remark 1 for why Property (6) is not expressible.] Again, it might come as a surprise that (3) is in fact expressible in the XPath query language[2]. This suggests that the resemblance to XPath, whose expressivity issues have received a lot of attention in databases [21, 22, 32, 33, 31], is again not as straightforward.

In recent years there have been a lot of work on static analysis and optimisation/minimisation of CSS styling rules in HTML applications, e.g., [12, 24, 6, 23, 14, 29]. Such work has been motivated by the well-known issues of CSS bloat — typically caused by HTML plugins with generic style sheets and CSS preprocessors, to name a few — whose amount has been estimated to be around 60% on average for typical industrial applications [24]. The latest W3C Recommendation of CSS Selectors Level 3 [7] gives an informal, natural language specification (more than 30 pages in printed form), which is not easy to work with when developing static analysis and optimisation tools for CSS. For this reason, researchers have attempted to develop concise formalisations of the CSS selector language, e.g., in μ-calculus [12].

Despite the amount of work in developing practical static analysis and optimisation tools for CSS, many basic foundational issues for static analysis of CSS have remained unsolved. Perhaps the simplest and the most fundamental static analysis problem for CSS concerns the decidability of *satisfiability* (a.k.a. *non-emptiness*) of CSS3 selectors: given a CSS selector, check whether it is *satisfiable* (i.e. the selector may match some node in some tree). If decidable, what is its computational complexity? Satisfiability often forms a basis of other static analysis problems. One such static analysis problem that is of immediate practical importance concerns the *non-disjointness* (a.k.a. *intersection*) problem for the CSS3 selectors: given two CSS selectors, decide whether they can simultaneously select the same node on some tree. Decidability and complexity for the intersection problem for CSS is currently not known. A solution to this problem is required to be able to perform CSS file minimisation [23] via the reordering and merging of styling rules (see Section 2 for an example). In particular, if two styling rules match the same node, the order of declarations in a file can determine the appearance of the document. To be able to safely reorder two rules, one must be sure that the selectors of the two rules will never match the same node in a tree. Existing solutions (e.g. [23]) only overapproximate the order dependency relation. A more precise relation would allow more CSS rules to be reordered (and therefore optimised).

Finally, we shall remark that these questions are not only of theoretical interest, but they are also useful for actual static analysis tools, which currently cannot handle the CSS selector

[2]XPath has a slightly different document object model in comparison to HTML. This statement is true if “class” is substituted by “tag”

language *in its entirety* (e.g. [6] handle $\sim 70\%$ of the selectors from their benchmarks). Part of the problem seems to be that hitherto there is no formalisation of CSS3 selectors as specified in the W3C Recommendation [7]. For example, the tree logic defined in [12] can express Property (3) above, which is not expressible as a CSS3 selector, but did not incorporate constraints on attributes (i.e. strings).

Contributions. In this paper we present a 1.5-page formalisation of the CSS3 selector language. Unlike previous formalisations, our formalisation attempt captures CSS3 *in its entirety*, as specified in the latest W3C Recommendation [7]. In particular, our formalisation models CSS3 constraints on attributes (i.e. strings), which yields semantics of CSS selectors as a set of finite "data trees" (i.e. whose node labels range over an infinite set). Equipped with the new formalisation, we then study the two aforementioned fundamental static analysis problems for CSS3: (1) satisfiability (a.k.a. non-emptiness) of CSS3 selectors, and (2) non-disjointness (a.k.a. non-emptiness-of-intersection, or simply "intersection") of two CSS3 selectors. We show that both problems are decidable and, in fact, NP-complete. This is in contrast to the previous formalisation of CSS selectors as a tree logic (which do not cover CSS3 in its entirety), in which both problems would be EXPTIME-complete.

With a very simple set of navigational axes (i.e. up, left, but neither down nor right), it might be a surprise that CSS satisfiability is NP-hard especially in the light of the polynomial-time solvability result for satisfiability for the XPath fragment[3] that uses only this set of navigational axes, e.g., see [15]. It turns out that one cause of this NP-hardness is CSS selectors' ability to do modulo arithmetic, conjunctions, and restricted negations; these can be used to encode the satisfiability of intersections of negations of arithmetic progressions, which is known to be NP-complete [30].

For the upper bound, we show that both problems can be reduced in polynomial-time to satisfiability of existential Presburger formulas, for which highly optimised SMT solvers are available. Our result suggests the potential application of SMT solvers for providing an efficient solution for static analysis of CSS3 selectors. Several tricky features of CSS3 that have to be handled in this reduction includes constraints for counting the number of children (which could express the existence of exponentially many children in the tree), built-in regular string matching constraints for the attribute values, uniqueness of id attributes in the tree, conjunctions, and restricted negations. A careful analysis of the language is required in pinpointing the exact computational complexity. For example, if CSS3 were to allow unrestricted unrestricted regular string matching constraints, it would be able to encode intersection of regular languages, which is PSPACE-complete [16].

Related Work. Although there has not been much work on the foundation of CSS, a lot of fundamental work exists on its cousin XPath. The first formalisation of XPath was given by Wadler [34]. Database researchers then developed several the *core logical* versions of the language that allows them to study expressiveness and static analysis issues in a cleaner setting, e.g., see [21, 5, 13, 32, 33, 31, 22]. To obtain a cleaner version of XPath, one often makes the assumption that there are only finitely many attribute values, which is not true for XPath in its entirety. XPath is also known to share many similarities to logics and automata over unranked trees, e.g., see [18] for a readable survey on this. Many results on the decidability and complexity of static analysis of XPath in this setting (with or without DTDs, and possibly

[3]XPath fragments that are commonly studied in database theory often assume a finite set of attribute values, no global id attribute constraints (i.e. no two nodes share the same id), and no counting constraints on the number of children, which is not the case for CSS3.

with some navigational directions disallowed) are available, e.g., see [15, 4, 26, 10, 20, 11]. Note that the static analysis problems for CSS that we consider in this paper do not come with a schema (e.g. in DTD), which is also the case for most existing work in static analysis and optimisation/minimisation on CSS [12, 24, 6, 23, 14, 29]. As we previously mentioned, if one were to make a similar simplification (e.g. finitely many attributes, no global id constraints, and no modular counting constraints on the number of children), CSS would be an XPath fragment in the database setting with only upward and leftward navigational combinators, which is known to be polynomial-time solvable [15]. This is in contrast to our NP-completeness for satisfiability for CSS3 selectors. The simplified setting of XML has recently been extended (e.g. see [9]) to models that allow reasoning of attribute values (from an abstract infinite domain), but this mostly results in models whose satisfiability problems exhibit high computational complexity (at least EXPTIME-hard) since they allow general comparisons of data values.

We also mention the work on automata on unranked trees (labels in the trees from a finite alphabet) that allow counting constraints on the number of children, e.g., see [28]. Such work typically allow counting constraints that are much more complex than what CSS3 allow, which is the reason for that checking emptiness for such automata is PSPACE-complete.

Organisation. Section 2 provides a CSS primer. We fix mathematical notation and terminologies that we use throughout the paper in Section 3. Section 4 provides a formalisation of CSS3 selectors. In Section 5 we provide an overview of results on satisfiability and intersection of CSS3 selectors and, in particular, proofs of the NP lower bounds. Section 6 and 7 develop the machinery for proving an NP upper bound for these problems. We conclude with future work in Section 8. See Appendix for omitted details.

2 CSS By Example

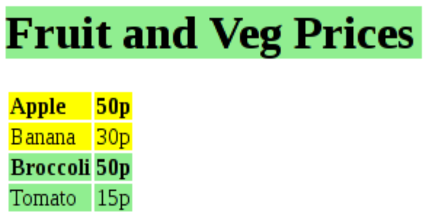
We give a brief overview of CSS selectors and motivate the problems studied in this paper by means of an example. We refer the readers to [2] for a gentle and more complete introduction to CSS.

A very simple web page is given in Figure 1. In this section we describe both HTML and CSS. The page contents are given by the HTML document shown in Figure 2. In Figure 3 we give the CSS file used to style the page. Although our example uses HTML, we note that CSS can also be used to identify nodes in XML trees.

2.1 Basics

HTML documents are written in a markup language that gives a tree structure. Each node in the tree is an element e. The description of the node begins with a tag $\langle e \rangle$ and ends with a tag $\langle /e \rangle$. Nodes appearing between these tags describe the children. As can be seen in Figure 2 the root of the tree is an HTML element containing two children: a head and a body. The `head` element contains metadata about the page, while the `body` element contains the page contents. The `h1` element is the heading, while the `table` element contains the prices. Each row of the table is a `tr` element, and each data item is a `td` element.

The opening tag of some of these elements contains additional information. E.g. `id=‘‘prices’’` and `class=‘‘fruit’’`. These are attributes that label a node. There are many possible attributes, but the id and class attributes have a special purpose for CSS selectors, as will be seen below. Note the table row containing tomatoes has two class values: `fruit` and `vegetable`.



Fruit and Veg Prices

Apple	**50p**
Banana	30p
Broccoli	**50p**
Tomato	15p

Figure 1: An example web page.

```
<html>
    <head><title>POPL Example</title></head>
    <body>
        <h1>Fruit and Veg Prices</h1>
        <table id="prices">
            <tbody>
                <tr class="fruit">
                    <td class="name">Apple</td>
                    <td class="price">50p</td>
                </tr>
                <tr class="fruit">
                    <td class="name">Banana</td>
                    <td class="price">30p</td>
                </tr>
                <tr class="vegetable">
                    <td class="name">Broccoli</td>
                    <td class="price">50p</td>
                </tr>
                <tr class="fruit vegetable">
                    <td class="name">Tomato</td>
                    <td class="price">15p</td>
                </tr>
            </tbody>
        </table>
    </body>
</html>
```

Figure 2: The HTML DOM Tree of the web page in Figure 1

The CSS stylesheet given in Figure 3 is used to customise its appearance. The stylesheet is a list of rules, each having the form

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

where *selectors* is a comma-separated list of selectors and *declarations* is a semi-colon separated list of (property) declarations. Selectors are used to identify nodes in the document tree. A rule is applied to a node if one of its selectors matches the node. Each declaration in a rule specifies how some aspect of the node is displayed.

For example, the simplest rule is the first

```
h1 { background: lightgreen }
```

In this rule the selector identifies any h1 element. The declaration specifies that the background should be light green. Hence, the heading of the webpage has a light green background.

The next rule

```
h1 { background: lightgreen }
.fruit { background: yellow }
.vegetable { background: lightgreen }
table#prices tr:nth-child(2n) { font-weight: normal }
table#prices tr:nth-child(2n+1) { font-weight: bold }
```

Figure 3: A stylesheet for the webpage in Figure 1.

```
.fruit { background: yellow }
```

has a selector that identifies any node whose class attribute contains "fruit". The syntax "." is used to specify classes. However, this is just syntactic sugar. CSS allows selectors to specify the contents of arbitrary attributes. The above rule could have been alternatively written

```
[class ~= fruit] { background: yellow }
```

where `~=` is an operator specifying that the attribute is a space-separated list of words, one of which is the value on the right. Similarly, #ι asserts that the id attribute is ι, or, equivalently, [id = ι].

The selector

```
table#prices tr:nth-child(2n+1)
```

demonstrates several more complex features of CSS. The first thing to notice is that the selector is split into two parts: table#prices and tr `:nth-child(2n + 1)`. These two parts put constraints on two separate nodes. The node identified by the selector is the node matched by tr `:nth-child(2n + 1)`. The whitespace between the two parts indicates that the second (matched) node should be a descendent of the first. CSS also allows the operators >, +, and ~ to indicate a child-of, neighbour-of, and sibling-of relationship respectively.

The first part table#prices is a conjunction, asserting that the first node is *both* a `table` element and has the ID "prices". The second part tr `:nth-child(2n + 1)` is also a conjunction. It asserts that the matched element is a `tr` element *and* satisfies `:nth-child(2n + 1)`. The `:nth-child(2n + 1)` asserts that the node is the ιth child of its parent such that $\iota = 2n+1$ for some integer value n. [The leftmost child is the *first*, instead of zeroth, child of the parent node.] Thus, the stylesheet in Figure 3 renders even rows of the table with a normal font weight, and odd rows in bold.

2.2 Static Analysis

In this section we discuss applications of the intersection problem. Consider from Figure 3. Readers familiar with CSS might be tempted to minimise the CSS in Figure 3 by replacing the first three rules with

```
h1, .vegetable { background: lightgreen }
.fruit { background: yellow }
```

Here, the first rule has a comma-separated list of selectors that identify which nodes should have a light green background. Merging the rules like this avoids the declaration "`background: lightgreen`" appearing twice, reducing the size of the file. However, such a transformation will change the appearance of the page. In particular, the tomato row will display in yellow rather than green. This is because the tomato row matches both .fruit and .vegetable. In cases when such a conflict arises, the CSS specification first uses a notion called *specificity* – which is a measure easily calculated from a selector's syntax that we will not discuss here.

In this case both selectors have the same specificity so the rule appearing latest in the file is used. The optimisation proposed above changes the order of the rules in the file and as a result changes the appearance. Hence, before attempting to minimise a file by merging related rules, we must first check whether selectors may overlap. Note, since industrial CSS files may contain thousands of rules, potentially millions of (pairwise) comparisons must be performed. Hence, it is important to pinpoint the precise complexity of the problem.

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. 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^+).

3.2 Trees

A *tree domain* is a set $D \subseteq (\mathbb{N}_{>0})^*$ of *nodes* that is both *prefix-closed* and *younger-sibling closed*. That is $\eta\iota \in D$ implies both $\eta \in D$, and $\eta\iota' \in D$ for all $\iota' < \iota$.

A Σ-*labelled tree* is a pair $T = (D, \lambda)$ where D is a tree domain, and $\lambda : D \to \Sigma$ 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 younger sibling of η', i.e., there is some η'' and $\iota \in \mathbb{N}_{>0}$ such that $\eta = \eta''\iota$ and $\eta' = \eta''(\iota - 1)$. We write $\eta \sim \eta'$ if η is a younger 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 $\eta' = \eta''\iota$. Finally, ϵ is the root node in any tree.

3.3 Existential Presburger Arithmetic

Recall that Presburger arithmetic is a first-order theory for reasoning about integer linear arithmetic [17]. In the sequel, we will only use the existential fragment of the theory (a.k.a. *existential Presburger arithmetic*), which is well-known to be NP-complete [27]. *Formulas* θ of existential Presburger arithmetic have the form

$$\theta, \theta' = \mathcal{E} \sim \mathcal{E} \;\big|\; \exists x.\theta \;\big|\; \theta \wedge \theta' \;\big|\; \theta \vee \theta'$$

where $\sim \in \{=, >, \geq, <, \leq\}$ and $\mathcal{E}$ are *(integer) expressions* $\mathcal{E}$ of the form $a_0 + a_1x_1 + \cdots + a_nx_n$ where each a_i is an integer (with a binary representation) and each x_i denotes an (integer) variable. We assume a standard semantics of Presburger arithmetic. Note, the existential operator $\exists$ quantifies over non-negative integers $\mathbb{N}$. We will implicitly assume that all free variables are existentially quantified. Moreover, we will occasionally allow quantification over a finite set of elements. It is straightforward to encode this into a quantification over integers of a bounded range.

4 Formalisation of DOM and CSS

In this section we give a formal definition of document trees and CSS3 selectors. Our formalisation follows the latest version of W3C recommendation of CSS3 selectors [7].

4.1 Definition of Document Trees

In this section we introduce Documents Objects Models (DOMs), which we also refer to as document trees. A document consists of a number of elements, which in turn may have sub-elements as children. Each node in the tree is given a *type* consisting of an element name and a namespace. For example an element `P` in the `HTML` namespace is a paragraph in an HTML document. In addition each node may be labelled by attributes. Each attribute is identified by a namespace (which may differ from that of the node) and an attribute name. The attributes take string values. Finally, a node may be labelled by a number of *pseudo-classes* which specify properties of the node in the document. For example `:enabled` means that the node is enabled and the user may interact with it, while `:empty` indicates that the node has no children. The set of pseudo-classes is fixed by the CSS specification.

In the formal definition below we permit a possibly infinite set of namespace, element, and attribute names. This is because the document writer can use any string of characters as a name. Thus, the sets of possible names cannot be fixed to a finite set 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 *elements* E, a possibly infinite set of *attribute names* A, and a possibly infinite alphabet[4] 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 to an attribute value (string), and one or more of the pseudo-classes. For a function $f_A \in \mathcal{F}_{\mathrm{fin}}(\mathrm{NS} \times A, \Gamma^*)$ we say $f_A(s, a) = \bot$ when f_A is undefined over $s \in \mathrm{NS}$ and $a \in A$, where $\bot \notin \Gamma^*$ 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.

We will write $s{:}e$ for an element e with namespace s in $\mathrm{NS} \times E$ and $s{:}a$ for an attribute a with namespace s in $\mathrm{NS} \times A$. For convenience, when $\lambda(\eta) = (s, e, f_A, P)$ we write

$$\lambda_{\mathtt{S}}(\eta) = s, \quad \lambda_{\mathtt{E}}(\eta) = e, \quad \lambda_{\mathtt{A}}(\eta) = f_A, \quad \lambda_{\mathtt{P}}(\eta) = P .$$

There are several consistency constraints on the pseudo-classes labelling a node.

- For all $s \in \mathrm{NS}$ 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`.

[4]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 is labelled `:empty` iff it has no children.

In the sequel, we will tacitly assume that document trees satisfy these consistency constraints.

We write $\text{Trees}(\text{NS}, E, A, \Gamma)$ for the set of such trees.

Note, readers familiar with HTML may have expected clauses asserting, for example, that if an node matches `:hover`, then its parent and "label controls" 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`.

Finally, note that CSS selectors can be applied to any document tree, e.g. XML or HTML. The semantics of CSS3 from the CSS3 selector specification [7] may allow document formats that are forbidden by the HTML5 specification.

4.2 Definition of CSS3

In the following sections we define CSS syntax and semantics. In formally, as CSS selector consists of *node selectors* σ – which match individual nodes in the tree – combined using the operators $\gg$, `>`, `+`, and `~`. These operators express the descendant-of, child-of, neighbour-of, and sibling-of relations respectively. Note, we use slightly different syntax to their counterpart semantical operators $\gg$, $>$, $+$, and $\sim$ in order to distinguish syntax from meaning.

A node selector σ has the form $\tau\Theta$ where τ constrains the *type* of the node. That is, τ places restrictions on the namespace and element labels of the node. The rest of the selector is a set Θ of *simple* selectors that assert atomic properties of the node. These may take several forms.

Class and ID selectors $.v$ and $\#v$ assert that the node has a class or ID v respectively. Note, a node may have several classes (given as a string by a space-separated list of classes) but only one ID.

The next type of simple selector is the attribute selectors that generally take the form $[s\,|\,a\ op\ v]$ for some namespace s, attribute a, operator $op \in \{$`=`, `~=`, `|=`, `^=`, `$=`, `*=`$\}$, and some string $v \in \Gamma^*$. The namespace may be omitted if the specification is agnostic about the namespace of the attribute. The operators `=`, `^=`, `$=`, and `*=` take their meaning from regular expressions. That is, equals, begins-with, ends-with, and contains respectively. The remaining operators are more subtle. The `~=` 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. en-GB means "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.

Attribute selectors may also take the form $[s\,|\,a]$ or $[a]$ to assert that the attribute is merely defined on the node.

Next, simple selectors may specify which pseudo-classes label a node. E.g. the selector `:enabled` ensures the node is currently enabled in the document.

There are several further kinds of pseudo-classes. Of particular interest are selectors such as `:nth-child(`α`n + `β`)`. These 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 $(2n + 1)$th node in the sibling order. That is, the node is at an odd position.

Finally, simple selectors may be negations `:not(`θ`)` of a simple selector θ with the condition that negations cannot be nested.

4.2.1 Syntax

Fix NS, E, A, and Γ as in the previous section. 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 ψ defined as:

$$\begin{array}{rcl} \psi & ::= & \varphi \mid \varphi\texttt{::first-line} \mid \varphi\texttt{::first-letter} \mid \\ & & \varphi\texttt{::before} \mid \varphi\texttt{::after} \end{array}$$

where

$$\varphi ::= \sigma \mid \varphi \gg \sigma \mid \varphi > \sigma \mid \varphi + \sigma \mid \varphi \sim \sigma \mid$$

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

$$\tau ::= \texttt{*} \mid \texttt{(}s\texttt{|*)} \mid e \mid \texttt{(}s\texttt{|}e\texttt{)}$$

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

$$\theta ::= \theta_\neg \mid \texttt{:not(}\sigma_\neg\texttt{)}$$

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

$$\sigma_\neg ::= \texttt{*} \mid \texttt{(}s\texttt{|*)} \mid e \mid \texttt{(}s\texttt{|}e\texttt{)} \mid \theta_\neg$$

and $\theta_\neg =$

$$\begin{array}{l} \texttt{.}v \mid \texttt{\#}v \mid \\ \texttt{[}a\texttt{]} \mid \texttt{[}a\texttt{ = }v\texttt{]} \mid \texttt{[}a\texttt{ ~= }v\texttt{]} \mid \texttt{[}a\texttt{ |= }v\texttt{]} \mid \\ \texttt{[}a\texttt{ ^= }v\texttt{]} \mid \texttt{[}a\texttt{ \$= }v\texttt{]} \mid \texttt{[}a\texttt{ *= }v\texttt{]} \mid \\ \texttt{[}s\texttt{|}a\texttt{]} \mid \texttt{[}s\texttt{|}a\texttt{ = }v\texttt{]} \mid \texttt{[}s\texttt{|}a\texttt{ ~= }v\texttt{]} \mid \texttt{[}s\texttt{|}a\texttt{ |= }v\texttt{]} \mid \\ \texttt{[}s\texttt{|}a\texttt{ ^= }v\texttt{]} \mid \texttt{[}s\texttt{|}a\texttt{ \$= }v\texttt{]} \mid \texttt{[}s\texttt{|}a\texttt{ *= }v\texttt{]} \mid \\ \texttt{:link} \mid \texttt{:visited} \mid \texttt{:hover} \mid \texttt{:active} \mid \texttt{:focus} \mid \\ \texttt{:enabled} \mid \texttt{:disabled} \mid \texttt{:checked} \mid \\ \texttt{:root} \mid \texttt{:empty} \mid \texttt{:target} \mid \\ \texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)} \mid \texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)} \mid \\ \texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)} \mid \texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)} \\ \texttt{:only-child} \mid \texttt{:only-of-type} \end{array}$$

with $s \in$ NS, $e \in E$, $a \in A$, $v \in \Gamma^*$, and $\alpha, \beta \in \mathbb{Z}$. Note, we will omit Θ when is it empty.

4.2.2 Removing 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.

$$\begin{array}{c} \varphi \mid \varphi\texttt{::first-line} \mid \varphi\texttt{::first-letter} \mid \\ \varphi\texttt{::before} \mid \varphi\texttt{::after} \end{array}$$

We are interested here in the nodes matched by a selector. The pseudo-elements `::first-line`, `::first-letter`, `::bef` 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 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.

4.2.3 Semantics

We now define the semantics of CSS3 selectors. 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 is defined inductively with respect to a document tree $T = (D, \lambda)$ and a node $\eta \in D$ as follows:

$$\begin{array}{lcllcl}
T, \eta & \models & \varphi \gg \sigma & \overset{\text{def}}{\Leftrightarrow} & \exists \eta' \gg \eta \;.\; (T, \eta' \models \varphi) \wedge (T, \eta \models \sigma) \\
T, \eta & \models & \varphi > \sigma & \overset{\text{def}}{\Leftrightarrow} & \exists \eta' > \eta \;.\; (T, \eta' \models \varphi) \wedge (T, \eta \models \sigma) \\
T, \eta & \models & \varphi + \sigma & \overset{\text{def}}{\Leftrightarrow} & \exists \eta' + \eta \;.\; (T, \eta' \models \varphi) \wedge (T, \eta \models \sigma) \\
T, \eta & \models & \varphi \,\tilde{}\, \sigma & \overset{\text{def}}{\Leftrightarrow} & \exists \eta' \sim \eta \;.\; (T, \eta' \models \varphi) \wedge (T, \eta \models \sigma)
\end{array}$$

and for node selectors we have

$$\begin{array}{rcllcl}
T, \eta & \models & \tau\Theta & \overset{\text{def}}{\Leftrightarrow} & (T, \eta \models \tau) \wedge \forall \theta \in \Theta \;.\; (T, \eta \models \theta) \\
T, \eta & \models & \texttt{*} & \overset{\text{def}}{\Leftrightarrow} & \top \\
T, \eta & \models & \texttt{(}s\texttt{|*)} & \overset{\text{def}}{\Leftrightarrow} & s = \lambda_{\texttt{S}}(\eta) \\
T, \eta & \models & e & \overset{\text{def}}{\Leftrightarrow} & e = \lambda_{\texttt{E}}(\eta) \\
T, \eta & \models & \texttt{(}s\texttt{|}e\texttt{)} & \overset{\text{def}}{\Leftrightarrow} & s = \lambda_{\texttt{S}}(\eta) \wedge e = \lambda_{\texttt{E}}(\eta) \\
\forall p \in P \;.\; T, \eta & \models & p & \overset{\text{def}}{\Leftrightarrow} & p \in \lambda_{\texttt{P}}(\eta)
\end{array}$$

and, for the remaining selectors we have (noting vv' is the concatenation of the strings v and

v' and v-v' is the concatenation of v and v' with a "-" in between)

$$
\begin{array}{lcll}
T, \eta & \models & \texttt{:not(}\theta_\neg\texttt{)} & \overset{\text{def}}{\Leftrightarrow} \quad \neg\,(T, \eta \models \theta_\neg) \\
T, \eta & \models & \texttt{.}v & \overset{\text{def}}{\Leftrightarrow} \quad \exists s \in \text{NS} . T, \eta \models \texttt{[}s\texttt{|class ~= }v\texttt{]} \\
T, \eta & \models & \texttt{\#}v & \overset{\text{def}}{\Leftrightarrow} \quad \exists s \in \text{NS} . T, \eta \models \texttt{[}s\texttt{|id = }v\texttt{]} \\
T, \eta & \models & \texttt{[}a\texttt{]} & \overset{\text{def}}{\Leftrightarrow} \quad \exists s \in \text{NS} . \; T, \eta \models \texttt{[}s\texttt{|}a\texttt{]} \\
T, \eta & \models & \texttt{[}a \; op \; v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} \quad \exists s \in \text{NS} . \; T, \eta \models \texttt{[}s\texttt{|}a \; op \; v\texttt{]} \\
T, \eta & \models & \texttt{[}s\texttt{|}a\texttt{]} & \overset{\text{def}}{\Leftrightarrow} \quad \lambda_{\texttt{A}}(\eta)(s, a) \neq \bot \\
T, \eta & \models & \texttt{[}s\texttt{|}a \texttt{ = } v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} \quad \lambda_{\texttt{A}}(\eta)(s, a) = v \\
T, \eta & \models & \texttt{[}s\texttt{|}a \texttt{ |= } v\texttt{]} & \overset{\text{def}}{\Leftrightarrow} \quad \left(\begin{array}{c} (\lambda_{\texttt{A}}(\eta)(s, a) = v) \; \vee \\ \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} \quad \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} \quad \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} \quad \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 $v\llcorner v'$ is the concatenation of v and v' with the space character $\llcorner$ in between)

$$
\begin{array}{c}
T, \eta \models \texttt{[}s\texttt{|}a \texttt{ ~= } v\texttt{]} \\
\overset{\text{def}}{\Leftrightarrow} \\
\left(\begin{array}{c}
\lambda_{\texttt{A}}(\eta)(s, a) = v \; \vee \\
\exists v' . (\lambda_{\texttt{A}}(\eta)(s, a) = v\llcorner v') \; \vee \\
\exists v' . (\lambda_{\texttt{A}}(\eta)(s, a) = v'\llcorner v) \; \vee \\
\exists v_1, v_2 . (\lambda_{\texttt{A}}(\eta)(s, a) = v_1\llcorner v\llcorner v_2)
\end{array} \right)
\end{array}
$$

then, for the counting selectors

$$T, \eta \models \texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)}$$
$$\overset{\text{def}}{\Leftrightarrow}$$
$$\exists n \, . \, \exists \eta', \iota \, . \, \left(\begin{array}{c} \eta = \eta'\iota \wedge \\ \iota = \alpha n + \beta \end{array} \right)$$

$$T, \eta \models \texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)}$$
$$\overset{\text{def}}{\Leftrightarrow}$$
$$\exists n \, . \, \exists \eta', \iota, \iota' \, . \, \left(\begin{array}{c} \eta = \eta'\iota \wedge \eta'\iota' \in D \wedge \\ \eta'(\iota' + 1) \notin D \wedge \\ \iota' - \iota + 1 = \alpha n + \beta \end{array} \right)$$

$$T, \eta \models \texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)}$$
$$\overset{\text{def}}{\Leftrightarrow}$$
$$\exists n \, . \, \exists \eta', \iota \, . \, \left(\left| \left\{ \eta'\iota' \;\middle|\; \begin{array}{c} \eta = \eta'\iota \wedge \\ \iota' \leq \iota \wedge \\ \lambda_{\mathtt{S}}(\eta'\iota') = \lambda_{\mathtt{S}}(\eta) \wedge \\ \lambda_{\mathtt{E}}(\eta'\iota') = \lambda_{\mathtt{E}}(\eta) \end{array} \right\} \right| = \alpha n + \beta \right)$$

$$T, \eta \models \texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)}$$
$$\overset{\text{def}}{\Leftrightarrow}$$
$$\exists n \, . \, \exists \eta', \iota \, . \, \left(\left| \left\{ \eta'\iota' \;\middle|\; \begin{array}{c} \eta = \eta'\iota \wedge \\ \iota' \geq \iota \; \wedge \eta'\iota' \in D \wedge \\ \lambda_{\mathtt{S}}(\eta'\iota') = \lambda_{\mathtt{S}}(\eta) \wedge \\ \lambda_{\mathtt{E}}(\eta'\iota') = \lambda_{\mathtt{E}}(\eta) \end{array} \right\} \right| = \alpha n + \beta \right)$$

Remark 1. *An interesting sidenote is that our formalisation allows one to prove certain inexpressibility results. A simple question is the following: what class of string languages can the attribute selectors not express? Answer: regular languages that are not* star-free *(i.e. can be described by regular expressions that do not allow Kleene star, but allow language complementation. A typical example of languages that are not star-free is $(aa)^*$, e.g., see [19]. To see this, one simply notes the attribute selectors can be translated to first-order formulas over strings (whose domains are string positions, whose unary predicates U_a can test the symbol at a given position is a, and whose binary predicate $\leq$ can test the precedence of two string positions).*

4.2.4 Divergences from full CSS

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

- Since class and ID selectors can be expressed in terms of attribute selectors (as in the semantics above), we will omit them in the remainder of the article.
- We assume each element name includes its 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[5] to `[`s`|lang |=` l`]`.

- We did not include `:indeterminate` since it was not formally introduced to CSS3.
- 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.
- 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 [3] (with a maximal possible codepoint) we feel it is reasonable to assume DOMs are also defined over a finite alphabet.

5 Static Analysis

Having formalised CSS3 selectors, we now consider basic static analsis problems for CSS3, especially non-emptiness (a.k.a. satisfiability) and intersection (a.k.a. non-disjointness) problems. Our main result is that both problems are NP-complete. The complexity holds whether or not numbers in CSS3 selectors are represented in unary or binary.

Theorem 1 (Non-Emptiness). *Given a CSS selector φ, deciding $\exists T, \eta\ .\ T, \eta \models \varphi$ is NP-complete.*

Theorem 2 (Intersection). *Given two CSS selectors φ_1 and φ_2, deciding $\exists T, \eta\ .\ T, \eta \models \varphi_1 \wedge T, \eta \models \varphi_2$ is NP-complete.*

We give the hardness results below. Membership is shown in the following sections. We introduce CSS automata for this purpose. In Proposition 5 we show each CSS selector can be translated in polynomial time into an equivalent CSS automaton. In Lemma 10 we show non-emptiness of CSS automata can be reduced to satisfiability of existential Presburger arithmetic, which is well-known to be in NP. This gives us membership of non-emptiness in NP. For intersection, we show in Proposition 6 that CSS automata are closed under intersection, and two automata can be intersected in polynomial time. Thus we reduce intersection in polynomial time to non-emptiness of a CSS automaton, which is in NP.

Lemma 3. *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 [30, 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

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

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 period 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

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

Thus, non-emptiness is NP-hard. □

Lemma 4. *Given two CSS selectors φ_1 and φ_2, deciding whether $\exists T, \eta \ . \ T, \eta \models \varphi_1 \wedge T, \eta \models \varphi_2$ is NP-hard.*

Proof. Non-emptiness reduces to non-emptiness-of-intersection. That is, φ is not empty iff the intersection of φ and `*` is non-empty. From Lemma 3 we get the result. □

6 CSS Automata

We will now develop a technique that allows us to simultaneously derive an NP upper bound for both the non-emptiness and the intersection problems for CSS3 selectors. Loosely speaking, these problems would be equivalent if CSS3 selectors were closed under intersection, which unfortunately are *not* due to the presence of CSS combinators in general. To this end, we follow a standard automata-theoretic approach: develop a general class of (tree) automata that subsume CSS3 selectors and are furthermore closed under intersection. We call these automata *CSS automata*. The result in this section shows that both non-emptiness and intersection problems for CSS3 selectors can be cast into the non-emptiness problem for CSS automata.

A CSS automaton navigates a tree structure in a similar manner to a CSS selector. That is, transitions may only move down the tree, or to a sibling. Furthermore, at each transition, a number of properties can be checked.

The reason we use automata as an intermediate step is that the automata navigate the tree in a uniform fashion. A direct intersection of two selectors is non-trivial since selectors may descend to different child nodes, and then meet again after the application of a number of sibling combinators. Thus their paths may diverge and combine several times. The automata are designed to always navigate to the first child, and then move from sibling to sibling. Thus, the intersection of CSS automata can be done with a straightforward product construction.

Intuitively, the modalities $\downarrow$, $\rightarrow$, $\rightarrow_+$, and $\circ$ perform the following operations. The $\downarrow$ operator moves to the first child of the current node. The $\rightarrow$ operator moves to the next sibling of the current node. Finally, the $\rightarrow_+$ operator moves an arbitrary number of siblings to the right.

Note, we only allow self loops in the automata, since CSS does not have loops. Loops in the automata are used to skip over nodes that are not matched by a node selector in the full selector (e.g. $.v$ ~ $.v'$ matches two sibling nodes with classes v and v' respectively; loops will be used to skip over all nodes appearing between these two matched nodes). We do not allow $\rightarrow$ to label a loop – this is for the purposes of the NP proof: a sequence of transitions over a loop $\rightarrow$ can be more succinctly represented as a single loop transition labelled $\rightarrow_+$.

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. This is because a single $\rightarrow_+$ transition in one automaton may correspond to several transitions of the other. Hence, it is convenient to be able to use the $\rightarrow_+$ transition several times instead of splitting a single transition into several parts.

There is no analogue of $\rightarrow_+$ for $\downarrow$ because $\rightarrow_+$ is needed to handle selectors such as `:nth-child(`α`n + `β`)` which enforce the existence of a number of preceding children. There is no such CSS selector to count the depth of the tree. Finally $\circ$ marks the node matched by the selector.

Definition 1 (CSS Automata)**.** *A CSS Automaton $\mathcal{A}$ is a tuple $\left(Q, \Delta, q^{in}, q_f\right)$ where Q is a finite set of states, $\Delta \subseteq Q \times \{\downarrow, \rightarrow, \rightarrow_+, \circ\} \times NSel \times Q$ is a transition relation, $q^{in} \in Q$ 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 $\sigma = *$, and*
3. *($\rightarrow$ doesn't label loops) for all $(q, d, \sigma, q) \in \Delta$ we have $d \neq \rightarrow$ and $\sigma = *$.*
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 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 $\mathcal{A}$, written $(T, \eta) \in \mathcal{L}(\mathcal{A})$, if there exists a sequence

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

such that

1. $q_0 = q^{\mathrm{in}}$ is the initial state and $\eta_0 = \varepsilon$ is the root node,
2. $q_{\ell+1} = q_f$ and $\eta_\ell = \eta$,
3. for all i, there is some transition $q_i \xrightarrow[\sigma]{d} q_{i+1}$ with η_i satisfying σ and if $i \leq \ell$,
   (a) if $d = \downarrow$ then $\eta_{i+1} = \eta_i 1$, (i.e., the youngest child of η_i)
   (b) if $d = \rightarrow$ then there is some η' and ι such that $\eta_i = \eta'\iota$ and $\eta_{i+1} = \eta'(\iota + 1)$, and
   (c) if $d = \rightarrow_+$ then there is some η', ι and ι' such that $\eta_i = \eta'\iota$ and $\eta_{i+1} = \eta'\iota'$ and $\iota' > \iota$.

6.1 CSS Selectors to CSS Automata

For each CSS selector φ we can construct a CSS automaton $\mathcal{A}_\varphi$ such that for all trees T

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

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 $o_i \in \{\gg, >, +, \sim\}$. We define

$$\mathcal{A}_\varphi = \left(Q, E, \Delta, q^{\mathrm{in}}, q_f\right)$$

where

$$\begin{array}{rcl} Q & = & \{\circ_i, \bullet_i \mid 1 \leq i \leq n\} \uplus \{q_f\} \\ q^{\mathrm{in}} & = & \circ_1 \end{array}$$

and we define the transition relation Δ to contain the following transitions, which 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.

$$\circ_1 \xrightarrow[*]{\downarrow, \rightarrow_+} \circ_1 \qquad \text{and} \qquad \circ_n \xrightarrow[\sigma_n]{\circ} q_f$$

and for each $1 \leq i < n$,

- if o_i is `>` then we have

$$\begin{tikzcd}
\circ_i \arrow[r, "\downarrow", "\sigma_i"'] \arrow[rr, bend left=50, "\downarrow", "\sigma_i"'] & \bullet_i \arrow[r, "\rightarrow", "*"'] \arrow[loop below, "\rightarrow_+"', "*"] & \circ_{i+1} \quad ,
\end{tikzcd}$$

  In other words, the automaton moves to the first child of the current node and move to the right zero or more steps[6].
- if o_i is `>>` then we have

$$\begin{tikzcd}
\circ_i \arrow[r, "\downarrow", "\sigma_i"'] \arrow[rr, bend left=50, "\downarrow", "\sigma_i"'] & \bullet_i \arrow[r, "{\rightarrow,\downarrow}", "*"'] \arrow[loop below, "{\rightarrow_+,\downarrow}"', "*"] & \circ_{i+1} \quad ,
\end{tikzcd}$$

  In other words, the automaton traverses the tree downward and rightward any number of steps.
- if o_i is `+` then we move to the next sibling:

$$\begin{tikzcd}
\circ_i \arrow[r, "\rightarrow", "\sigma_i"'] & \circ_{i+1} \quad ,
\end{tikzcd}$$

- if o_i is `~` then we have

$$\begin{tikzcd}
\circ_i \arrow[r, "\rightarrow", "\sigma_i"'] \arrow[rr, bend left=50, "\rightarrow", "\sigma_i"'] & \bullet_i \arrow[r, "\rightarrow", "*"'] \arrow[loop below, "\rightarrow_+"', "*"] & \circ_{i+1} \quad .
\end{tikzcd}$$

We prove the following in Lemma 11 (soundness) and Lemma 12 (completeness) in Appendix A.1.

Proposition 5. *For each CSS selector φ and tree T, we have $T, \eta \models \varphi$ iff $(T, \eta) \in \mathcal{L}(\mathcal{A}_\varphi)$.*

6.2 Intersection of CSS Automata

Intersection of CSS automata is a standard product construction where $\rightarrow_+$ can match a sequence of $\rightarrow$ transitions. We first define the intersection of two node selectors. Note node selectors are of the form $\tau\Theta$ where $\tau \in \{*, (s|*), e, (s|e) \mid s \in \text{NS} \wedge e \in E\}$. Hence, letting

[6]Recall $\rightarrow_+$ appears on loops only as this eases the intersection construction later.

$\Theta = \Theta_1 \cup \Theta_2$. we define

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

We give the product construction required for intersection below. Note that the synchronisation of a transition labelled $\rightarrow$ with a transition labelled $\rightarrow_+$ relies on $\rightarrow_+$ only labelling loops[7].

Definition 2 ($\mathcal{A}_1 \cap \mathcal{A}_2$). *Given* $\mathcal{A}_1 = \left(Q_1, E, \Delta_1, q_1^{in}, q_f^1\right)$ *and* $\mathcal{A}_2 = \left(Q_2, E, \Delta_2, q_2^{in}, q_f^2\right)$ *we define*

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

where $\Delta =$

$$\begin{array}{l} \left\{ (q_1, q_2) \xrightarrow[\sigma_1 \cap \sigma_2]{\downarrow} (q_1', q_2') \;\middle|\; q_1 \xrightarrow[\sigma_1]{\downarrow} q_1' \wedge q_2 \xrightarrow[\sigma_2]{\downarrow} q_2' \right\} \cup \\ \left\{ (q_1, q_2) \xrightarrow[\sigma_1 \cap \sigma_2]{\rightarrow} (q_1', q_2') \;\middle|\; q_1 \xrightarrow[\sigma_1]{\rightarrow} q_1' \wedge q_2 \xrightarrow[\sigma_2]{\rightarrow} q_2' \right\} \cup \\ \left\{ (q_1, q_2) \xrightarrow[\sigma_1]{\rightarrow} (q_1', q_2) \;\middle|\; q_1 \xrightarrow[\sigma_1]{\rightarrow} q_1' \wedge q_2 \xrightarrow[\texttt{*}]{\rightarrow_+} q_2 \right\} \cup \\ \left\{ (q_1, q_2) \xrightarrow[\sigma_2]{\rightarrow} (q_1, q_2') \;\middle|\; q_1 \xrightarrow[\texttt{*}]{\rightarrow_+} q_1 \wedge q_2 \xrightarrow[\sigma_2]{\rightarrow} q_2' \right\} \cup \\ \left\{ (q_1, q_2) \xrightarrow[\texttt{*}]{\rightarrow_+} (q_1, q_2) \;\middle|\; q_1 \xrightarrow[\texttt{*}]{\rightarrow_+} q_1 \wedge q_2 \xrightarrow[\texttt{*}]{\rightarrow_+} q_2 \right\} \cup \\ \left\{ (q_1, q_2) \xrightarrow[\sigma_1 \cap \sigma_2]{\circ} \left(q_f^1, q_f^2\right) \;\middle|\; q_1 \xrightarrow[\sigma_1]{\circ} q_f^1 \wedge q_2 \xrightarrow[\sigma_2]{\circ} q_f^2 \right\} . \end{array}$$

The following proposition asserts correctness of the above construction, which we prove in Appendix A.2.

Proposition 6. *Given two CSS automata* $\mathcal{A}_1$ *and* $\mathcal{A}_2$*, we have for all trees* T *and nodes* η*,*

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

7 Non-Emptiness Algorithm

We show non-emptiness of a CSS automaton can be decided in NP by a polynomial-time reduction to existential Presburger arithmetic. That is, given a CSS automaton $\mathcal{A}$, our algorithm

[7]Else we would have to deal with the case where $\rightarrow_+$ could read several nodes while $\rightarrow$ can only read one.

constructs an existential Presburger formula $\theta_{\mathcal{A}}$ such that $\mathcal{A}$ accepts a non-empty language iff $\theta_{\mathcal{A}}$ is satisfiable. Our existential Presburger encoding relies on the fact that any loop in the automaton is a self loop that only needs to be taken at most once. In the case of 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 $\rightarrow_+$, it may be that selectors such as `:nth-child(`α`n + `β`)` enforce the existence of a number of intermediate nodes. However, 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.

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.

7.1 Bounding Namespaces and Elements

Before giving the details of the reduction to existential Presburger, we first show that the number of namespaces and elements can be bounded linearly in the size of the automaton.

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 $\sigma = (s \,|\, e)\Theta$ for some s, e, and Θ,
2. $S_{\mathcal{A}}$ is the set of transitions $q \xrightarrow[\sigma]{d} q' \in \Delta$ with $\sigma \neq *$ 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.

We prove the following proposition in Appendix B.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 7 (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 $(T', \eta) \in \mathcal{L}(\mathcal{A})$ where $T' = (D, \lambda')$ and that the image of λ' contains only namespaced elements in $\downharpoonleft(E_{\mathcal{A}})$.*

Thus, we can define bounded sets of namespaces and elements

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

7.2 Encoding Attribute Selectors

When encoding non-emptiness checks we will need to encode the satisfiability of conjunctions of attribute selectors. We show here how to perform this encoding. First we collect attribute selectors that apply to individual attributes. Next, we argue that if a collection of selectors can be satisfied by an attribute value, then there is a value of polynomially bounded length. Using this bound, we can encode the selectors as an existential Presburger formula.

7.2.1 Polynomial Bound

We begin by arguing the existence of a polynomial bound for the solutions to any finite set C of constraints of the form `[`s`|`a 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.

$$\begin{tikzcd}
q^{\text{in}} \arrow[r, bend left, "\overset{\rightarrow}{\texttt{[}s\texttt{|id = }v\texttt{]}}"] \arrow[r, bend right, "\underset{\texttt{[}s\texttt{|id = }v\texttt{]}}{\downarrow}"'] & q \arrow[r, "\circ", "*"'] & q_f
\end{tikzcd}$$

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 8 (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 [25]. 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 $\text{Prefs}(C)$ be the set of words v' such that v' is a prefix of some v with $\texttt{[}s\texttt{|}a\ op\ v\texttt{]} \in C$ or $\texttt{:not([s|a}\ op\ \texttt{v])} \in C$. Moreover, let ^ 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 3 ($\mathcal{A}_C$)**.** *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*
  - *$v \in \text{Prefs}(C)$, 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 $v\$$) $S_i =$

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

and $V_i =$

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

Such a run is *accepting* if $S_n = \{\texttt{[}s\texttt{|}a\ op\ v\texttt{]} \mid \texttt{[}s\texttt{|}a\ op\ v\texttt{]} \in C\}$ 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 in Appendix B.2. 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 9 (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 8 (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 $N'_c \leq N_c$.

7.2.2 Presburger Encoding

Since short witnessing strings suffice for satisfiable attribute attribute selectors, we may use quantifier-free Presburger formulas to "guess" these witnessing strings; the letter in each position in a witnessing string is encoded by a number. [In fact, boolean formulas suffice, though we shall use integer variables since they are cleaner to work with and do not yield worse computational complexity.]

In the following encoding, we use the fact 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. Note, these fresh namespaces do not appear in $\rfloor(\mathrm{NS})$.

Let *op* range over {`=`, `~=`, `|=`, `^=`, `$=`, `*=`}. 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`|`a`]` or `[`s`|`a 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 sets of selectors is presented in the sequel. Let $\mathrm{Neg}(s, a) = \{$`:not([s|a` *op* `v])` $|$ `:not([a` *op* `v])` $\in \Theta\}$.

- For each s and a with Θ^s_a non-empty and containing at least one selector of the form `[`s`|`a`]` or `[`s`|`a op v`]`, we enforce

$$\mathrm{AttsPres}_{s:a}(\Theta^s_a \cup \mathrm{Neg}(s, a), i)$$

  if `:not([a])`, `:not([s|a])` $\notin \Theta$, else we assert false.
- For each `[`a`]` $\in \Theta$, let s be fresh namespace. We assert

$$\mathrm{AttsPres}_{s:a}(\{\texttt{[}s\texttt{|}a\texttt{]}\} \cup \mathrm{Neg}(s, a), i)$$

  and for each `[`a op v`]` $\in \Theta$ we assert

$$\mathrm{AttsPres}_{s:a}(\{\texttt{[}s\texttt{|}a\ op\ v\texttt{]}\} \cup \mathrm{Neg}(s, a), i)$$

  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. We obtain this by, for each s and a, collecting the sets of constraints over s and a from each transition, and applying Lemma 8 (Bounded Attribute Values). We then take the maximum of all bounds obtained in this way[8]. 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 for each character position of the satisfying value, and encode the constraints almost directly. That

[8]Of course, we could obtain individual bounds for each s and a if we wanted to streamline the encoding.

is, letting θ range over positive attribute selectors, we define[9]

$$\begin{array}{rcl} \text{AttsPres}_{s:a}(C, i) & = & \bigwedge\limits_{\theta \in C} \text{AttsPres}(\theta, \vec{x}) \wedge \\ & & \bigwedge\limits_{\texttt{:not(}\theta\texttt{)} \in C} \neg\text{AttsPres}(\theta, \vec{x}) \wedge \\ & & \text{Nulls}(\vec{x}) \end{array}$$

where $\vec{x} = x^{s:a}_{i,1}, \ldots, x^{s:a}_{i,N}$ will be existentially quantified later in the encoding and whose values will range[10] over $\Gamma \uplus \{0\}$ where 0 is a null character used to pad the suffix of each word. We define $\text{AttsPres}(\theta, \vec{x})$ for several θ, the rest can be defined in the same way (see Appendix B.3). Letting $v = a_1 \ldots a_m$,

$$\begin{array}{rcl} \text{AttsPres}(\texttt{[}s\texttt{|}a\texttt{]}, \vec{x}) & = & \top \\ \text{AttsPres}(\texttt{[}s\texttt{|}a \texttt{ = } v\texttt{]}, \vec{x}) & = & \bigwedge\limits_{1 \leq i \leq m} x^{s:a}_{i,j} = a_j \wedge x^{s:a}_{i,m+1} = 0 \\ \text{AttsPres}(\texttt{[}s\texttt{|}a \texttt{ ^= } v\texttt{]}, \vec{x}) & = & \bigwedge\limits_{1 \leq j \leq m} x^{s:a}_{i,j} = a_j \\ \text{AttsPres}(\texttt{[}s\texttt{|}a \texttt{ *= } v\texttt{]}, \vec{x}) & = & \bigvee\limits_{0 \leq j \leq N-m-1} \bigwedge\limits_{1 \leq j' \leq m} x^{s:a}_{i,j+j'} = a_j \end{array}$$

Finally, we enforce correct use of the null character

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

7.3 Encoding Non-Emptiness

We encode non-emptiness using the following variables for $0 \leq i \leq n$, representing the node at the ith step of the run:

- $\overline{q}_i$, 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 $\downharpoonleft(\text{NS})$ indicating the element tag (with namespace) of the ith node read in the run,
- $\overline{e}_i$, taking any value in $\downharpoonleft(E)$ 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
- $\overline{n}^{s:e}_i$, for all $s \in \downharpoonleft(\text{NS})$ and $e \in \downharpoonleft(E)$, taking a natural number variable indicating that there are $\overline{n}^{s:e}_i$ 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
- $\overline{N}^{s:e}_i$, for all $s \in \downharpoonleft(\text{NS})$ and $e \in \downharpoonleft(E)$, taking a natural number variable indicating that there are $\overline{N}^{s:e}_i$ nodes of type $s{:}e$ strictly following the current node in the sibling order, and

[9]Note, we allow negation in this formula. This is for convenience only as the formulas we negate can easily be transformed into existential Presburger.

[10]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.

- $x^{s:a}_{i,j}$ 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[11].

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

Before we give the reduction for CSS automaton we first show how to negate a formula of the form $\exists \overline{n}.\overline{x} = \alpha\overline{n} + \beta$ where $\overline{x}$ is a variable, and α and β are constants, while still remaining inside existential Presburger arithmetic. This is not difficult, but one has to take care since the numbers α and β are represented in binary. Then we show how to construct an existential Presburger formula from a node selector, and a position i.

7.3.1 Negating Positional Formulas

We need formulas to negate selectors like `:nth-child(`α`n + `β`)`, For completeness, we give the definition of the negation below. We prove its correctness in Appendix B.4.

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 $\beta_2 \leq 0$ and $\beta_1\alpha > 0$ implies $\beta_2 \geq 0$.

Definition 4 (NoMatch$(\overline{x}, \alpha, \beta)$)**.** *Given constants α, β, β_1, and β_2 as above, we define* NoMatch$(\overline{x}, \alpha, \beta)$ *to be*

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

7.3.2 Encoding Node Selectors

We define the encoding of node selectors below. This encoding uses 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.

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

Definition 5 (Pres(σ, i))**.** *Given a node selector $\tau\Theta$, we define*

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

[11]Recall s is not necessarily in $\downarrow$(NS) as it may be some fresh value.

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

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

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 $i > 0$

$$\mathrm{Pres}(\texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)}, i) = \exists \overline{n}.\overline{n}_i = \alpha\overline{n}+\beta$$

$$\mathrm{Pres}(\texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)}, i) = \exists \overline{n}.\overline{N}_i = \alpha\overline{n}+\beta$$

$$\mathrm{Pres}(\texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)}, i) = \bigvee_{\substack{s \in \downarrow(\mathrm{NS}) \\ e \in \downarrow(E)}} \left(\begin{array}{c} \overline{s}_i = s \wedge \overline{e}_i = e \ \wedge \\ \exists \overline{n}.\overline{n}_i^{s:e} + 1 = \alpha\overline{n} + \beta \end{array} \right)$$

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

$$\mathrm{Pres}(\texttt{:only-child}, i) = \overline{n}_i = 1 \wedge \overline{N}_i = 1$$

$$\mathrm{Pres}(\texttt{:only-of-type}, i) = \bigvee_{\substack{s \in \downharpoonleft(\mathrm{NS}) \\ e \in \downharpoonleft(E)}} \begin{pmatrix} \overline{s}_i = s \wedge \overline{e}_i = e \wedge \\ \overline{n}_i^{s:e} = 0 \wedge \overline{N}_i^{s:e} = 0 \end{pmatrix}$$

and we define the translation of negated simple CSS selectors as

$$\begin{aligned}
\mathrm{Pres}_\neg(\texttt{*}, i) &= \bot \\
\mathrm{Pres}_\neg((s\texttt{|*}), i) &= (\overline{s}_i \neq s) \\
\mathrm{Pres}_\neg(e, i) &= (\overline{e}_i \neq e) \\
\mathrm{Pres}_\neg((s\texttt{|}e), i) &= (\overline{s}_i \neq s \vee \overline{e}_i \neq e) \\
\mathrm{Pres}_\neg(\texttt{:root}, i) &= \begin{cases} \bot & i = 0 \\ \top & \textit{otherwise} \end{cases} \\
\forall p \in P \setminus \{\texttt{:root}\}\ .\ \mathrm{Pres}_\neg(p, i) &= \neg\overline{p}_i
\end{aligned}$$

and finally, for the remaining selectors

$$\begin{aligned}
\mathrm{Pres}_\neg(\texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)}, 0) &= \top \\
\mathrm{Pres}_\neg(\texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)}, 0) &= \top \\
\mathrm{Pres}_\neg(\texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)}, 0) &= \top \\
\mathrm{Pres}_\neg(\texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)}, 0) &= \top \\
\mathrm{Pres}_\neg(\texttt{:only-child}, 0) &= \top \\
\mathrm{Pres}_\neg(\texttt{:only-of-type}, 0) &= \top
\end{aligned}$$

and when $i > 0$

$$\mathrm{Pres}_\neg(\texttt{:nth-child(}\alpha\texttt{n + }\beta\texttt{)}, i) = \mathrm{NoMatch}(\overline{n}_i, \alpha, \beta)$$

$$\mathrm{Pres}_\neg(\texttt{:nth-last-child(}\alpha\texttt{n + }\beta\texttt{)}, i) = \mathrm{NoMatch}\left(\overline{N}_i, \alpha, \beta\right)$$

$$\mathrm{Pres}_\neg(\texttt{:nth-of-type(}\alpha\texttt{n + }\beta\texttt{)}, i) = \bigvee_{\substack{s \in \downharpoonleft(\mathrm{NS}) \\ e \in \downharpoonleft(E)}} \begin{pmatrix} \overline{s}_i = s \wedge \overline{e}_i = e \wedge \\ \mathrm{NoMatch}(\overline{n}_i^{s:e} + 1, \alpha, \beta) \end{pmatrix}$$

$$\mathrm{Pres}_\neg(\texttt{:nth-last-of-type(}\alpha\texttt{n + }\beta\texttt{)}, i) = \bigvee_{\substack{s \in \downharpoonleft(\mathrm{NS}) \\ e \in \downharpoonleft(E)}} \begin{pmatrix} \overline{s}_i = s \wedge \overline{e}_i = e \wedge \\ \mathrm{NoMatch}\left(\overline{N}_i^{s:e} + 1, \alpha, \beta\right) \end{pmatrix}$$

$$\text{Pres}_\neg(\texttt{:only-child}, i) = \overline{n}_i > 1 \vee \overline{N}_i > 1$$

$$\text{Pres}_\neg(\texttt{:only-of-type}, i) = \bigvee_{\substack{s \in \downharpoonleft(\text{NS}) \\ e \in \downharpoonleft(E)}} \left(\begin{array}{c} \overline{s}_i = s \wedge \overline{e}_i = e \wedge \\ \overline{n}_i^{s:e} > 0 \vee \overline{N}_i^{s:e} > 0 \end{array} \right) .$$

7.3.3 Reducing CSS Automata

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 $0 \leq i \leq n$. We encode non-emptiness of a CSS automaton using the encoding below.

Definition 6. $\theta_\mathcal{A}$ *Given a CSS automaton* $\mathcal{A}$ *we define*

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

where Tran(i) *and* Consistent *are defined below.*

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

We define $\text{Tran}(i) = \bigvee_{t \in \Delta} \text{Tran}(i, t)$ where $\text{Tran}(i, t)$ is defined below by cases. 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 $\text{Tran}(i, t)$ to be

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

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

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

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

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

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

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

The consistency constraint on the run asserts

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

where each conjunct is defined below.

- The clause $\mathrm{Consistent}_n$ asserts that the values of $\overline{n}_i$, $\overline{N}_i$, $\overline{n}^{s:e}_i$, and $\overline{N}^{s:e}_i$ are consistent. That is
$$\bigwedge_{1 \leq i \leq n} \left(\overline{n}_i = 1 + \sum_{s:e \in E} \overline{n}^{s:e}_i \right) \wedge \left(\overline{N}_i = 1 + \sum_{s:e \in E} \overline{N}^{s:e}_i \right) \ .$$
- 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 $x^{s:\mathtt{id}}_{i,j}$ we assert
$$\bigwedge_{1 \leq i \neq i' \leq n} \ \bigvee_{1 \leq j \leq N} x^{s:\mathtt{id}}_{i,j} \neq x^{s:\mathtt{id}}_{i',j} \ .$$
- Finally, $\mathrm{Consistent}_p$ asserts the remaining consistency constraints on the pseudo-classes. We define $\mathrm{Consistent}_p =$
$$\bigwedge_{0 \leq i \leq n} \left(\begin{array}{c}
\neg \left(\overline{\mathtt{:link}}_i \wedge \overline{\mathtt{:visited}}_i \right) \ \wedge \\
\bigwedge\limits_{0 \leq j \neq i \leq n} \left(\ \neg \left(\overline{\mathtt{:target}}_i \wedge \overline{\mathtt{:target}}_j \right) \ \right) \ \wedge \\
\neg \left(\overline{\mathtt{:enabled}}_i \wedge \overline{\mathtt{:disabled}}_i \right)
\end{array} \right) .$$

  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.

Lemma 10 (Correctness of $\theta_{\mathcal{A}}$)**.** *For a CSS automaton $\mathcal{A}$, we have*

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

Proof. See Lemma 14 and Lemma 15 in Appendix B.5. □

8 Conclusion and Future Work

This paper presents a formalisation of CSS3 selectors and fundamental results concerning static analysis of CSS3 selectors. In particular, we chose to study the problems of satisfiability and disjointness of CSS3 selectors since they are arguably the most basic static analysis problems for CSS3 selectors and they have found applications in CSS optimisation. To the best of our knowledge, we are the first to successfully argue that these static analysis problems are decidable and, in fact, NP-complete. Our reduction to existential Presburger formulas (for which highly optimised SMT solvers are available) could potentially lead to efficient solutions to these static analysis problems that can handle CSS3 in its entirety. We leave this implementation task for future work.

Another future research avenue is to study other fundamental static analysis and optimisation problems for CSS3. One such problem is the problem of inclusion-checking for CSS selectors (which have found applications in CSS optimisation; see [6]). It is possible to use the technique developed in this paper to show that the problem is decidable, though it is far from obvious what the actual computational complexity is. Finally, since CSS is an evolving language (e.g. see [8] for the latest draft for the currently unstable CSS4, but seems to be substantially more expressive than CSS3), we believe that academics could play an important role in providing constructive feedback on the expressiveness of the language. Our formalisation of CSS3 is a starting point. We leave it for future work to address this issue in more depth.

References

[1] W3C Recommendation of XPath 3.0 (referred in June 2016). `https://www.w3.org/TR/xpath-30/`.

[2] W3Schools Gentle Introduction to CSS (referred in June 2016). `http://www.w3schools.com/css/css3_intro.asp`.

[3] The Unicode Standard, Version 6.0 (referred in June 2016). `http://www.unicode.org/versions/Unicode6.0.0`.

[4] Michael Benedikt, Wenfei Fan, and Floris Geerts. 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*, pages 25–36, 2005.

[5] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, pages 79–95, 2003.

[6] Martí Bosch, Pierre Genevès, and Nabil Layaïda. Reasoning with style. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2227–2233, 2015.

[7] T. Çelik, E. J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams. Selectors level 3: W3c recommendation 29 september 2011, 2011.

[8] Elika J. Etemad and Tab Atkins Jr. Selectors Level 4: W3C Working Draft 2 May 2013, 2013.

[9] Diego Figueira. *Reasoning on words and trees with data: On decidable automata on data words and data trees in relation to satisfiability of LTL and XPath.* PhD thesis, Ecole Normale Superieure de Cachan, 2010.

[10] Floris Geerts and Wenfei Fan. 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*, pages 122–137, 2005.

[11] Pierre Genevès and Nabil Layaïda. A system for the static analysis of XPath. *ACM Trans. Inf. Syst.*, 24(4):475–502, 2006.

[12] Pierre Genevès, Nabil Layaïda, and Vincent Quint. 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*, pages 809–818, 2012.

[13] Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 189–202, 2002.

[14] Matthew Hague, Anthony Widjaja Lin, and C.-H. Luke Ong. 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*, pages 1–19, 2015.

[15] Jan Hidders. Satisfiability of XPath expressions. In *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*, pages 21–36, 2003.

[16] Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 254–266, 1977.

[17] Dexter C. Kozen. *Theory of Computation.* Springer, 2006.

[18] Leonid Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

[19] Leonid Libkin. *Elements of Finite Model Theory.* Springer, 2010.

[20] Leonid Libkin and Cristina Sirangelo. Reasoning about XML with temporal logics and automata. *J. Applied Logic*, 8(2):210–232, 2010.

[21] Maarten Marx. Conditional xpath. *ACM Trans. Database Syst.*, 30(4):929–959, 2005.

[22] Maarten Marx and Maarten de Rijke. Semantic characterizations of navigational xpath. *SIGMOD Record*, 34(2):41–46, 2005.

[23] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. 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*, pages 496–506, 2014.

[24] Ali Mesbah and Shabnam Mirshokraie. Automated Analysis of CSS Rules to Support Style Maintenance. In *ICSE*, pages 408–418, 2012.

[25] A. Muscholl and I. Walukiewicz. An np-complete fragment of LTL. *Int. J. Found. Comput. Sci.*, 16(4):743–753, 2005.

[26] Frank Neven and Thomas Schwentick. On the complexity of xpath containment in the presence of disjunction, dtds, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[27] B. Scarpellini. Complexity of subcases of presburger arithmetic. *Trans. of AMS*, 284(1):203–218, 1984.

[28] Helmut Seidl, Thomas Schwentick, Anca Muscholl, and Peter Habermehl. Counting in trees for free. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 1136–1149, 2004.

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

[30] Larry J. Stockmeyer and Albert R. Meyer. 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*, pages 1–9, 1973.

[31] Balder ten Cate, Tadeusz Litak, and Maarten Marx. Complete axiomatizations for XPath fragments. *J. Applied Logic*, 8(2):153–172, 2010.

[32] Balder ten Cate and Maarten Marx. Navigational XPath: calculus and algebra. *SIGMOD Record*,

36(2):19–26, 2007.

[33] Balder ten Cate and Maarten Marx. Axiomatizing the logical core of XPath 2.0. *Theory Comput. Syst.*, 44(4):561–589, 2009.

[34] Philip Wadler. Two semantics of XPath, 2000. Tech. rep., Bell Labs.

A Proofs for CSS Automata

A.1 Correctness of $\mathcal{A}_\varphi$

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

Lemma 11. *For each CSS selector φ and tree T, we have*

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

Proof. Suppose $(T, \eta) \in \mathcal{L}(\mathcal{A}_\varphi)$. 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 12. *For each CSS selector φ and tree T, we have*

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

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. $\square$

A.2 Correctness of Intersection

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

$$\left(q_1^1, q_1^2\right) \xrightarrow[\sigma_1]{d_1} \cdots \xrightarrow[\sigma_n]{d_n} \left(q_{n+1}^1, q_{n+1}^2\right)$$

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

- When $(\downarrow, \downarrow)$ we have $d_j = \downarrow$ and $\sigma_j = \sigma_j^1 \cap \sigma_j^2$.
- When $(\rightarrow, \rightarrow)$ we have $d_j = \rightarrow$ and $\sigma_j = \sigma_j^1 \cap \sigma_j^2$.
- When $(\rightarrow, \rightarrow_+)$ we have $d_j = \rightarrow$ and $\sigma_j = \sigma_j^1$.
- When $(\rightarrow_+, \rightarrow)$ we have $d_j = \rightarrow$ and $\sigma_j = \sigma_j^2$.
- When $(\rightarrow_+, \rightarrow_+)$ we have $d_j = \rightarrow_+$ and $\sigma_j = *$.
- When $(\circ, \circ)$ we have $d_j = \circ$ and $\sigma_j = \sigma_j^1 \cap \sigma_j^2$.
- 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_j^i iff it also satisfies σ_j. By observing $\sigma \cap * = * \cap \sigma = \sigma$ we always have $\sigma_j = \sigma_j^1 \cap \sigma_j^2$.

Let $\sigma_j^i = \tau_i \Theta_i$ 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 $\tau, \tau_1, \tau_2 \in \{*, (s|*), (s|e), e\}$ else the type selectors cannot be satisfied (either $\tau =$ `:not(*)` or τ_1 and τ_2 assert conflicting namespaces or elements).

If some $\tau_i = *$ 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 $\tau = (s|e)$ and $\tau_1 = (s|*)$ 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.

B Proofs for Non-Emptiness of CSS Automata

B.1 Bounding Namespaces and Elements

We show Proposition 7 (Bounded Types). 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 $\mathit{Nodes}_{s:e}$ be the set of nodes labelled by $s{:}e$ in λ. In λ' we label all nodes in $\mathit{Nodes}_{s:e}$ by $s{:}e$. That is, we do not relabel nodes labelled by $s{:}e$. Let Nodes be the union of all such $\mathit{Nodes}_{s:e}$.

Next we consider all $0 \leq i \leq \ell$ such that $\eta_i \notin \mathit{Nodes}$ (i.e. was not labelled in the previous case) and $t_i = q_i \xrightarrow[\sigma]{d} q_{i+1}$ with $\sigma \neq *$. 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 $\mathit{Nodes}_{s:e}$ (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`|*)`, or `(`s`|`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.

B.2 Proof of Lemma 9 (Short Attribute Values)

We give the proof of Lemma 9. 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.

B.3 Missing definitions for $\mathrm{AttsPres}(\theta, \vec{x})$

$$
\begin{array}{l}
\mathrm{AttsPres}(\texttt{[}s\,\texttt{|}\,a\ \texttt{\~{}=}\ v\texttt{]}, \vec{x}) = \\
\displaystyle\bigwedge_{1 \le j \le m} \left(\begin{array}{c} x^{s:a}_{i,j} = a_j \ \wedge \\ \left(\begin{array}{c} x^{s:a}_{i,m+1} = 0 \\ \vee \\ x^{s:a}_{i,m+1} = \text{\textvisiblespace} \end{array} \right) \end{array} \right) \vee \\
\displaystyle\bigvee_{1 \le j \le N-m-1} \left(\begin{array}{c} x^{s:a}_{i,j-1} = \text{\textvisiblespace} \ \wedge \\ \displaystyle\bigwedge_{1 \le j' \le m} x^{s:a}_{i,j+j'} = a_j \ \wedge \\ \left(x^{s:a}_{i,j+m+1} = 0 \vee x^{s:a}_{i,j+m+1} = \text{\textvisiblespace} \right) \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{AttsPres}(\texttt{[}s\,\texttt{|}\,a\ \texttt{|=}\ v\texttt{]}, \vec{x}) = \\
\displaystyle\bigwedge_{1 \le j \le m} x^{s:a}_{i,j} = a_j \wedge \left(x^{s:a}_{i,m+1} = 0 \vee x^{s:a}_{i,m+1} = \texttt{-} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{AttsPres}(\texttt{[}s\,\texttt{|}\,a\ \texttt{\$=}\ v\texttt{]}, \vec{x}) = \\
\displaystyle\bigvee_{0 \le j \le N-m-1} \left(\begin{array}{c} \displaystyle\bigwedge_{1 \le j' \le m} x^{s:a}_{i,j+j'} = a_j \ \wedge \\ x^{s:a}_{i,j+m+1} = 0 \end{array} \right)
\end{array}
$$

B.4 Correctness of $\mathrm{NoMatch}(\overline{x}, \alpha, \beta)$

We show that our negation of periodic constraints is correct.

Proposition 13 (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 $|\beta'_2| < 0$ 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 $0 < |\beta_2 - \beta'_2| < |\alpha|$.

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 $\text{NoMatch}(\overline{x}, \alpha, \beta)$ 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 $0 < |\beta_2 - \beta_2'| < |\alpha|$.

In the other direction, we have three cases depending on the satisfied disjunct of $\text{NoMatch}(\overline{x}, \alpha, \beta)$. 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 $0 < |\beta_2 - \beta_2'| < |\alpha|$. Hence, there can be no $\overline{n}$ with $\overline{x} = \alpha\overline{n} + \alpha\beta_1 + \beta_2$. □

B.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 14. *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

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

that is an accepting run. We know from Proposition 7 (Bounded Types) that T can only use namespaces from $\downharpoonleft(\text{NS})$ and elements from $\downharpoonleft(E)$. 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 $i > \ell + 1$.
- $\overline{s}_i = \lambda_{\mathtt{S}}(\eta_i)$ for all $i \leq \ell + 1$ ($\overline{s}_i$ can take any value for other values of i).
- $\overline{e}_i = \lambda_{\mathtt{E}}(\eta_i)$ for all $i \leq \ell + 1$ ($\overline{e}_i$ can take any value for other values of i).
- $\overline{p}_i = (p \in \lambda_{\mathtt{P}}(\eta_i))$, 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

$$j = \left| \left\{ \eta'\iota' \;\middle|\; \begin{array}{c} \iota' < \iota \;\wedge \eta'\iota' \in D \wedge \\ \lambda_{\mathtt{S}}(\eta'\iota') = \lambda_{\mathtt{S}}(\eta) \wedge \\ \lambda_{\mathtt{E}}(\eta'\iota') = \lambda_{\mathtt{E}}(\eta) \end{array} \right\} \right| .$$

  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

$$j = \left| \left\{ \eta'\iota' \;\middle|\; \begin{array}{c} \iota' > \iota \;\wedge \eta'\iota' \in D \wedge \\ \lambda_{\mathtt{S}}(\eta'\iota') = \lambda_{\mathtt{S}}(\eta) \wedge \\ \lambda_{\mathtt{E}}(\eta'\iota') = \lambda_{\mathtt{E}}(\eta) \end{array} \right\} \right| .$$

  When $i = 0$ or $i > \ell + 1$ we can assign any value to $\overline{N}_i^{s:e}$.

- Assignments to $x^{s:a}_{i,j}$ are discussed below.

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

Recall

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

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 $\neg\overline{\texttt{:empty}}_i$ since we know $\texttt{:empty} \notin \lambda_{\mathtt{P}}(\eta_i)$. 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

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

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 `:nth-child(`α`n + `β`)` and `:nth-last-child(`α`n + `β`)` satisfaction of the encoding follows immediately from η satisfying θ and our assignment to $\overline{n}_i$ and $\overline{N}_i$. Similarly we satisfy encodings of `:nth-of-type(`α`n + `β`)`, `:nth-last-of-type(`α`n + `β`)`, `:only-child`, and `:only-of-type`. 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 thesame 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 13 (Correctness of NoMatch$(\overline{x}, \alpha, \beta)$). Similarly we satisfy encodings of `:not(:nth-of-type(`α`n + `β`))` and also of the selector `:not(:nth-last-of-type(`α`n + `β`))`. For `:not(:only-child)`, and `:not(:only-of-type)` the latter follow since an with multiple children the node must either not be position 1 from the start or end, and a not only of type node has more than 0 strict predecessors or successors of the same type.

Next, to satisfy AttsPres$(\tau\Theta, i)$ we have to satisfy a number of conjuncts. First, if we have a word $a_1 \dots a_n$ 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 AttsPres$_{s:a}$(`[`s`|`a op v`]`$, i$) or $\neg$AttsPres$_{s:a}$(`[`s`|`a op v`]`$, i$) is satisfied by our variable assignment. Similarly Nulls$(\vec{x})$ is straightforwardly satisfied. Hence, if a word satisfies C then our assignment to the variables means 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 $\lambda_{\mathtt{A}}(\eta)(s, a)$ 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 $\lambda_{\mathtt{A}}(\eta)(s, a)$ 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 $\lambda_{\mathtt{A}}(\eta)(s, a)$ (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 Consistent$_n$ above. The remaining consistency constraints are easily seen to be satisfied: Consistent$_i$ because each ID is unique causing at least one pair of characters to differ in every value; 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 15. *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 $q_0, \dots, q_{\ell+1}$ which is the prefix of the assignment to $\overline{q}_0, \dots, \overline{q}_n$ where q_ℓ is the first occurrence of q_f. We will construct a series of transitions $t_0, \dots, t_\ell$ 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.

- $\lambda_{\mathtt{S}}(\eta_i) = \rho(\overline{s}_i)$ (i.e. we assign the value given to $\overline{s}_i$ in the satisfying assignment).

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

We pick t_i as the transition corresponding to a satisfied disjunct of 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 $\lambda_{\mathtt{S}}(\eta') = s$, $\lambda_{\mathtt{E}}(\eta') = e$, $\lambda_{\mathtt{P}}(\eta') = \emptyset$, $\lambda_{\mathtt{A}}(\eta') = \emptyset$.
- 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 $\lambda_{\mathtt{S}}(\eta') = s$, $\lambda_{\mathtt{E}}(\eta') = e$, $\lambda_{\mathtt{P}}(\eta') = \emptyset$, $\lambda_{\mathtt{A}}(\eta') = \emptyset$. Note 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 Pres(τ, 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 Pres(τ, 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 $p \in \lambda_{\mathtt{P}}(\eta_i)$. 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 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 `:only-of-type` we know there are no other children of the same type since $\rho(\overline{n}^{s:e}_i) = \rho\left(\overline{N}^{s:e}_i\right) =$

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 $p \notin \lambda_{\mathtt{P}}(\eta_i)$. 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 Pres(`:not(:nth-child(`α`n + `β`))`$, i$) we obtain via Proposition 13 (Correctness of NoMatch$(\overline{x}, \alpha, \beta)$) that η_i does not satisfy the selector `:nth-child(`α`n + `β`)`. Similarly for the last child selector `:not(:nth-last-child(`α`n + `β`))`.

For the `:not(:nth-of-type(`α`n + `β`))` selector 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 13 (Correctness of 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 Consistent$_i$ and Consistent$_p$.

First, 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 Consistent$_p$. Thus, we are done. $\square$