On CSS Unsatisfiability Problem in the Presence of DTDs**

Nobutaka SUZUKI†, Member, Takuya OKADA††, and Yeondae KWON†††∗, Nonmembers

SUMMARY Cascading Style Sheets (CSS) is a popular language for describing the styles of XML documents as well as HTML documents. To resolve conflicts among CSS rules, CSS has a mechanism called specificity. For a DTD $D$ and a CSS code $R$, due to specificity $R$ may contain “unsatisfiable” rules under $D$, e.g., rules that are not applied to any element of any document valid for $D$. In this paper, we consider the problem of detecting unsatisfiable CSS rules under DTDs. We focus on CSS fragments in which descendant, child, adjacent sibling, and general sibling combinators are allowed. We show that the problem is coNP-hard in most cases, even if only one of the four combinators is allowed and under very restricted DTDs. We also show that the problem is in coNP or PSPACE depending on restrictions on DTDs and CSS. Finally, we present four conditions under which the problem can be solved in polynomial time.

key words: CSS, DTD, satisfiability

1. Introduction

Cascading Style Sheets (CSS) is a popular language for describing the styles of (X)HTML documents. CSS is also widely used as a stylesheet language for XML documents, e.g., DocBook [21] and MathML [2]. A CSS code is described by listing CSS rules that assign property values to elements. For example, the following CSS rule assigns property value serif to the font-family property of li elements that is a descendant of a ul element.

ul li {font-family:serif}

It is often the case that a CSS code contains conflicting CSS rules, i.e., CSS rules with the same property match the same HTML/XML element. To resolve such conflicts, the W3C CSS specification∗∗∗ provides a mechanism called specificity. In short, a CSS rule with a more specific selector has higher priority; if multiple CSS rules match the same HTML/XML element, then only the rule with the largest specificity is applied to the element.

In this paper, we consider CSS unsatisfiability problem, which is to detect unsatisfiable CSS rules under DTDs.

We say that a CSS rule $r$ is unsatisfiable under a DTD $D$ if $r$ is not applied to any element of any document valid for $D$, even if the selector of $r$ is “valid” for $D$. For example, consider the DTD and the CSS rules in Fig. 1. The second CSS rule “$c$ {font-family:sans-serif}” is unsatisfiable. To see this, consider the first and second CSS rules. The former matches a $c$ element that has an $a$ element as its ancestor, while the latter matches any $c$ element. Since the first CSS rule has more labels (“$a$” and “$c$”) than the second rule (“$c$”), the first CSS rule has a higher specificity. Moreover, under the DTD every $c$ element has an $a$ element as an ancestor. Hence the first CSS rule must be applied to any $c$ element, meaning that the second CSS rule is unsatisfiable. We have another unsatisfiable CSS rule; among the third, fourth, and fifth CSS rules, which have the same specificity and match an $f$ element, the third CSS rule “$a$ $f$ {font-family:serif}” is unsatisfiable due to the following reasons.

- In the fourth CSS rule ‘$>$’ denotes child selector, meaning that the rule matches an $f$ element having a $b$ element as its parent. Thus such an $f$ element can be matched by the fourth CSS rule as well as the third CSS rule. But only the fourth CSS rule is applied to such an $f$ element, since the latest rule takes the priority according to the CSS specification.
- In the fifth CSS rule ‘$+$’ denotes adjacent sibling selector, meaning that the rule matches an $f$ element having a $c$ element as its immediate left sibling. Because of a similar reason above, the fifth CSS rule, not the third CSS rule, is applied to such an $f$ element.
- Under the DTD, every $f$ element has a $b$ element as its parent or a $c$ element as its immediate left sibling.

\[
\begin{align*}
&\text{<!DOCTYPE a[} \\
&\quad \text{<ELEMENT a (bc)>} \\
&\quad \text{<ELEMENT b (fe)>} \\
&\quad \text{<ELEMENT c (#PCDATA)>} \\
&\quad \text{<ELEMENT e (cf)*>} \\
&\quad \text{<ELEMENT f (#PCDATA)>} \\
&\}] \\
&\text{a c {font-family:serif}} \\
&\text{c {font-family:sans-serif}} \\
&\text{a f {font-family:serif}} \\
&\text{f {font-family:sans-serif}} \\
\end{align*}
\]

**Fig. 1 Example of unsatisfiable CSS rules under DTD

*https://www.w3.org/TR/CSS22/

---

Manuscript received January 4, 2021.
Manuscript publicized March 10, 2021.
†The author is with University of Tsukuba, Tsukuba-shi, 305–8550 Japan.
††The author is with Nihon Unisys, Ltd, Tokyo, 135–8560 Japan.
†††The author is with University of Tokyo, Tokyo, 105–0003 Japan.
∗∗Presently, with Research Center for Agricultural Information Technology, National Agriculture and Food Research Organization.
∗∗∗The previous version of this paper is [20].
a) E-mail: nsuzuki@slis.tsukuba.ac.jp
DOI: 10.1587/transinf.2021EDP7002

Copyright © 2021 The Institute of Electronics, Information and Communication Engineers
As stated above, the fourth CSS rule is applied to the former and the fifth CSS rule is applied to the latter, meaning that there is no element to which the third CSS rule is applied.

Unsatisfiable CSS rules are clearly redundant, and thus reduce the readability of CSS codes, make the maintenance of CSS codes more difficult, and may slow the rendering speed of HTML/XML documents in browsers [14]. Therefore, unsatisfiable CSS rules should be detected and removed from CSS codes.

Although this problem seems to be easily solvable due to the simplicity of CSS rules, we show that the problem is intractable in most cases. We focus on simple CSS fragments in which selectors using whitespace ‘.’ (descendant), ‘>’ (child), ‘~’ (general sibling), and ‘+’ (adjacent sibling) as combinators are allowed (attribute selectors are omitted since attribute values are hard to predict from DTDs). The obtained results are summarized in Table 1. First, the problem is coNP-hard even if the DTD is disjunction-free and closure-free, and non-recursive DTD. Here, a DTD is duplicate-free [16] if each element name appears at most once in each content model. As an exceptional case, the problem is coNP-hard if only one of the four combinators is allowed. Furthermore, the problem is still coNP-hard under very restricted DTDs, e.g., closure-free, duplicate-free, and non-recursive DTDs. Here, a DTD is duplicate-free [16] if each element name appears at most once in each content model. As an exceptional case, the problem is in PTIME if the DTD is restricted to be disjunction-free and closure-free, and ‘+’ and ‘-’ are allowed as CSS combinators. These results imply that restricting the combinators of CSS rules or content models of DTDs does not effectively cope with the intractability of the problem. In this paper, we explore tractable cases of the problem from a different perspective and present four tractable cases (R1 to R4 in Table 1). In short, conditions R1 and R2 mean that, although the problem is intractable even if only ‘+’ or ‘-’ combinator is allowed, it becomes tractable by disabling universal selectors. R3 is a condition that restricts the number of conflicting CSS rules and imposes a slight restriction on their simple selectors. Finally, R4 is a condition that restricts the length of selectors, rather than restricting the number of conflicting rules.

### Related Work

Several methods for static analysis of CSS rules have been proposed. Geneves et al. proposed a logic-based system for analyzing CSS codes [7]. This system checks CSS rules by converting them into logic representations. Their encoding from CSS rules to logical formulae is linear, and their solver runs in EXPTIME. However, any other result on the complexity is not presented, and the unsatisfiability of CSS rules under DTDs is not discussed. Bosch et al. proposed a method for refactoring CSS rules [3], and Mazinan et al. also proposed methods for detecting duplicated CSS rules [13], [14]. These methods check for redundant CSS rules without DTDs, and complexity is not considered either. To the best of our knowledge, there have been no studies on the complexity of detecting unsatisfiable CSS rules in the presence of DTDs.

In addition to the above static analysis methods, a number of methods for analyzing CSS with HTML instances have been proposed. FireBug [5] and Chrome Developer Tools [9] are popular debugging tools that can detect CSS properties that are not applied to any elements of an HTML document. Hague et al. proposed a method for detecting redundant CSS rules in HTML5 applications [10]. Mesbah et al. proposed a method for detecting CSS rules that are unused in given HTML documents [15]. Practically, instance-level checking is different from our problem in the following sense. In websites and collections of XML documents, a CSS code is often shared by multiple documents. Furthermore, HTML/XML documents may be updated by adding/deleting/relabelling elements. Therefore, even if an instance-level checking detects a CSS rule that is not applied to any of the elements in a given HTML/XML document, the redundancy of the rule is still not obvious. In such cases, if we find out that the CSS rule is unsatisfiable, then we can safely remove it since it can never be used in any other documents sharing the CSS code or any updated versions of the document.

Another related problem is the XPath satisfiability.
problem. This problem is to decide, for an XPath expression \( p \) and a DTD \( D \), whether there exists an XML document \( d \) valid for \( D \) such that the answer of \( p \) on \( d \) is not empty. Benedikt et al. investigated the complexity of the problem, and showed that the problem is intractable in a number of cases [1]. To cope with this intractability, several restrictions on DTDs and XPath expressions have been proposed. Montazerian et al. showed that satisfiability of XPath expressions with child axes and qualifiers is tractable under duplicate-free DTDs [16]. Several other tractable XPath classes under duplicate-free DTDs were presented in our previous work [19]. Ishihara et al. proposed MRW-DTD [12] that is a class of DTDs broader than duplicate-free DTD, in which satisfiability of several practical XPath fragments is tractable. The major difference between the XPath satisfiability problem and our problem is as follows. In the former problem, complexity is determined by a single XPath expression, say \( p \), in that XPath expressions have no specificity and thus whether \( p \) is satisfiable is not affected by any other XPath expressions. On the other hand, in our problem complexity is not determined solely by a “target” CSS rule. Instead, it is determined by the interrelationship between a target CSS rule \( r \) and the CSS rules conflicting with \( r \).

Finally, a few studies on RDF query satisfiability have been made. Hartig proposed a model for Linked Data query processing and considered query satisfiability on the model [11]. Zhang et al. considered satisfiability of SPARQL pattern query and presented decidable classes of the satisfiability problem [22]. Both studies considered query satisfiability without any schema.

The rest of this paper is organized as follows. Section 2 provides some definitions related to CSS and DTDs. We also formalize the unsatisfiability problem. Section 3 presents the intractable cases of the problem. Section 4 describes four conditions under which the problem can be solved in polynomial time. Section 5 summarizes our conclusions.

2. Preliminaries

In this section, we firstly present some definitions related to tree, CSS, and DTDs, and then formalize the target problem.

2.1 Tree

An XML document is modeled as a rooted labeled tree. Formally, a rooted labeled tree (tree for short) is defined as follows.

- A single node \( v \) is a tree, where \( v \) is the root.
- Let \( v \) be a node and \( t_1, t_2, \ldots, t_n \) be trees with roots \( v_1, v_2, \ldots, v_n \), respectively. Then adding an edge from \( v \) to \( v_i \) for each \( 1 \leq i \leq n \) yields a tree rooted at \( v \).

Let \( \Sigma \) be a set of labels. Each node in a tree has a label in \( \Sigma \) and two nodes might have a same label. For a node \( v \) in a tree \( t \), \( l(v) \in \Sigma \) denotes the label of \( v \). For example, Fig. 2 depicts an XML document and its tree representation, where \( l(v_1) = \text{book}, l(v_2) = \text{title}, \) and so on. As shown in the figure, each element of the XML instance corresponds to a node in the tree.

2.2 CSS

A simple selector is either a label in \( \Sigma \) or a universal selector denoted ‘*’. Let \( t \) be a tree. A simple selector \( s \) matches a node \( v \) in \( t \) if \( s = \text{‘*’} \) or \( s \) is a label such that \( s = l(v) \).

A selector is a chain of one or more simple selectors connected by combinators, where a combinator is either a whitespace ‘\ ’ (descendant), ‘>’ (child), ‘~’ (general sibling), or ‘\ ’ (adjacent sibling). The length of a selector \( sel \), denoted \( \text{len}(sel) \), is the number of simple selectors in \( sel \). For example, if \( sel = a_1 \ast > c \), then \( \text{len}(sel) = 3 \).

Let \( s, s’ \) be simple selectors and \( (v, v’) \) be a pair of nodes in \( t \).

- A descendant selector \( s \ast s’ \) matches \( (v, v’) \) if \( s \) matches \( v, s’ \) matches \( v’ \), and \( v’ \) is a descendant of \( v \).
- A child selector \( s > s’ \) matches \( (v, v’) \) if \( s \) matches \( v, s’ \) matches \( v’ \), and \( v’ \) is a child of \( v \).
- A general sibling selector \( s \sim s’ \) matches \( (v, v’) \) if \( s \) matches \( v, s’ \) matches \( v’ \), and \( v’ \) is a (possibly non-immediate) right sibling of \( v \).
- An adjacent sibling selector \( s + s’ \) matches \( (v, v’) \) if \( s \) matches \( v, s’ \) matches \( v’ \), and \( v’ \) is the immediate right sibling of \( v \).

Let \( sel = s_1 c_1 s_2 c_2 \cdots s_{n-1} c_{n-1} s_n \) be a selector, where \( s_i \) is a simple selector and \( c_i \) is a combinator. Then \( sel \) matches a pair \( (v, v’) \) of nodes in \( t \) if

- \( n = 1 \), \( v = v’ \), and simple selector \( s_1 \) matches \( v \), or
- \( n > 1 \) and there exists a sequence \( v = v_1, v_2, \ldots, v_n = v’ \) of distinct nodes such that \( s_{i-1} c_{i-1} s_i \) matches \( (v_{i-1}, v_i) \) for every \( 2 \leq i \leq n \).

A CSS rule is denoted \( sel : p \vdash v \), where \( sel \) is a selector, \( p \) is a property, and \( v \) is a value.\(^1\) For a CSS rule \( r \), \( sel(r) \) denotes the selector of \( r \) and \( prop(r) \) denotes the property of \( r \). For a selector \( sel \), \( \text{first}(sel) \) denotes the first simple selector of \( sel \) and \( \text{last}(sel) \) denotes the last simple selector of \( sel \). For example, if \( r = a \ast b + c \vdash v \), then \( sel(r) = a \ast b + c,\) \( prop(r) = p,\) \( \text{first}(sel(r)) = a,\) and \( \text{last}(sel(r)) = c \). In the following, we assume that for any CSS rule \( r \), the last simple

---

\(^1\) A real CSS rule has a set of “property:value” declarations rather than a single declaration. However, such a CSS rule can be treated as CSS rules with a single declaration, e.g., \( sel \{ p_1 : v_1, p_2 : v_2 \} \) can be treated as two CSS rules \( sel p_1 : v_1 \) and \( sel p_2 : v_2 \).
selector \( last(\text{sel}(r)) \) is a label.

By \( \text{spec}(\text{sel}) \), we mean the number of labels occurring in selector \( \text{sel} \), e.g., if \( \text{sel} = a_+c \), then \( \text{spec}(\text{sel}) = 2 \). Since attributes are omitted in this paper, \( \text{spec}(\text{sel}) \) represents the specificity of \( \text{sel} \). A CSS code is defined as a list \( R \) of CSS rules. By \( \text{index}_R(\text{r}) \), we mean the index of a CSS rule \( \text{r} \) in \( R \), e.g., if \( R = [r, r’, r”] \), then \( \text{index}_R(r) = 1 \) and \( \text{index}_R(r”) = 3 \). We write \( r \in R \) if \( r \) occurs in \( R \). For a tree \( t \) and a node \( v \) in \( t \), a CSS rule \( r \in R \) is applied to \( v \) if

- for some node \( v’ \) in \( t \), \( \text{sel}(r) \) matches \( (v’, v) \), and
- for any CSS rule \( r’ \in R \) such that \( \text{sel}(r’) \) matches \( (v”, v) \) for some node \( v” \) and that \( \text{prop}(r) = \text{prop}(r’) \),
  (a) \( \text{spec}(\text{sel}(r)) > \text{spec}(\text{sel}(r’)) \) or
  (b) \( \text{spec}(\text{sel}(r)) = \text{spec}(\text{sel}(r’)) \) and \( \text{index}_R(r) > \text{index}_R(r’) \).

Let \( \text{CSS} \) be the set of CSS rules. We denote a fragment of \( \text{CSS} \) by listing the combinators supported by the fragment. For example, \( \text{CSS}^{(\omega \rightarrow)} \) denotes the set of CSS rules using only ‘\( \omega \)’ and ‘\( \rightarrow \)’ as combinators.

2.3 Unsatisfiability of CSS Rule under DTD

A DTD is a pair \( \text{DTD} = (d, s) \), where \( d \) is a mapping from \( \Sigma \) to the set of regular expressions over \( \Sigma \) and \( s \in \Sigma \) is the start label of \( d \). For a label \( a \in \Sigma \), \( d(a) \) is the content model of \( a \). For example, consider the following DTD.

```xml
<!DOCTYPE book[
  <!ELEMENT book (title, author+)>]
  <!ELEMENT title (PCDATA)>
  <!ELEMENT author (name)>]
```

Then the above DTD can be denoted by a pair \( (d, \text{book}) \), where \( d(\text{title}) = \text{title}\ast \), \( d(\text{author}) = \text{name}\ast \), and \( d(\text{title}) = \text{title}\ast \), \( d(\text{author}) = \text{name}\ast \). A tree is valid for \( D \) if (i) the root of \( t \) is labeled by \( s \) and (ii) for each node \( v \) in \( t \), \( l(t_1), l(t_2), \ldots, l(t_m), \ldots L(d(t(v))) \), where \( v_1, v_2, \ldots, v_m \) are the children of \( v \) and \( L(d(t(v))) \) is the language of \( d(t(v)) \).

Let \( R \) be a CSS code and \( r \in R \). If for any tree \( t \) valid for \( D \) and for any node \( v \) of \( t \), \( r \) is not applied to \( v \). Let \( C \) be a set of combinators. For a fragment \( \text{CSS}^C \), the CSS unsatisfiability problem, denoted \( \text{UNSAT} (\text{CSS}^C) \), is stated as follows.

**Input:** A DTD \( D = (d, a) \), a CSS code \( R \) of CSS rules in \( \text{CSS}^C \), and a CSS rule \( r \in R \)

**Problem:** Determine if \( r \) is unsatisfiable under \( D \) w.r.t. \( R \)

3. Intractability

In this section, we discuss intractable cases of the CSS unsatisfiability problem. We first consider the lower bound of complexity for the problem. We then consider the upper bound.

In the following discussion, we consider the following restricted DTDs as well as general (non-restricted) DTDs.

- For a regular expression \( e \), we say that \( e \) is *duplicate free* if each label in \( \Sigma \) occurs at most once in \( e \). A DTD \( D \) is *duplicate free* if each content model of \( D \) is duplicate free.
- For a regular expression \( e \), we say that \( e \) is *closure free* if \( e \) contains no Kleene closure (‘\( \ast \)’ and ‘\( + \)’). Then, a DTD \( D \) is *closure free* if each content model of \( D \) is closure free.
- For a DTD \( D = (d, s) \), \( b \) is reachable from \( a \) if (i) \( b \) occurs in \( d(a) \) or (ii) for some label \( c \), \( c \) is reachable from \( a \) and \( b \) occurs in \( d(c) \). We say that \( D \) is *non-recursive* if for any label \( a \), \( a \) is not reachable from \( a \) in \( D \).
- A DTD \( D \) is *disjunction-free* if any content model of \( D \) contains no disjunction ‘\( \lor \)’.

3.1 Lower Bound

We first consider the lower bound of complexity for the problem. As shown below, the problem is intractable even for very restricted DTDs and CSS rules. We first consider CSS rules using child/parent combinators, then consider CSS rules using sibling combinators.

**Theorem 1:** \( \text{UNSAT} (\text{CSS}^{(\omega \rightarrow)}) \) is coNP-hard under closure-free, duplicate-free, and non-recursive DTDs.

**Proof.** We reduce the 3DNF-tautology problem, which is coNP-complete [6], to the CSS unsatisfiability problem. The 3DNF-tautology problem is defined as follows.

**Input:** A 3DNF-formula \( \phi \)

**Problem:** Determine if \( \phi \) is a tautology, i.e., \( \phi \) is true for every truth assignment to \( \phi \)

Let \( \phi = (l_{11} \land l_{12} \land l_{13}) \lor (l_{21} \land l_{22} \land l_{23}) \lor \cdots \lor (l_{m1} \land l_{m2} \land l_{m3}) \) be a 3DNF formula, where \( l_j \) is a literal. Let \( \{x_1, x_2, \ldots, x_n\} \) be the set of variables of \( \phi \). Without loss of generality, we assume that literals in a clause are sorted by the indexes of variables, e.g., \( (x_1 \land \neg x_3 \land x_4) \) satisfies the assumption but \( (x_1 \land \neg x_4 \land \neg x_3) \) does not.

From the 3DNF-formula \( \phi \), we define a DTD \( D \), a CSS code \( R \) in which each rule uses only ‘\( \ast \)’ as a combinator, and a CSS rule \( r \in R \). First, the DTD \( D = (d, s) \) is defined as follows:

\[
\begin{align*}
d(s) &= X_1T|X_{1F},
d(X_{1T}) &= d(X_{1F}) = X_{2T}|X_{2F}, \\
d(X_{1F}) &= d(X_{2F}) = X_{3T}|X_{3F}, \\
& \vdots \\
d(X_{n-1T}) &= d(X_{n-1F}) = X_{nT}|X_{nF}, \\
d(X_{nT}) &= d(X_{nF}) = b, \\
d(b) &= e.
\end{align*}
\]

Here, \( X_{iT} \) is a label that represents “\( x_i \) is true” and \( X_{iF} \) is a label that represents “\( x_i \) is false”. Therefore, a tree valid for \( D \) represents a truth assignment to \( \phi \).
Next, we define the CSS code $R$. Let $R = [r_1, r_2, \ldots, r_m, r_B]$, where

\[
\begin{align*}
  r_i &= L_{i1} \land L_{i2} \land L_{i3} \land b \land p : v_i, \\
  r_B &= b \land p : v,
\end{align*}
\]

and $L_{ij}$ is a label defined as follows $(1 \leq i \leq m, 1 \leq j \leq 3)$:

\[
L_{ij} = \begin{cases} 
  X_{iT}, & \text{if } i_j = x, \\
  X_{iF}, & \text{if } i_j = \neg x
\end{cases}
\]

(1)

We show that $\phi$ is a tautology if and only if $r_B$ is unsatisfiable under $D$ w.r.t. $R$.

(⇒) Suppose that $\phi$ is a tautology. Then, for any truth assignment to $\phi$, there exists a term $(l_1 \land l_2 \land l_3)$ that becomes true. Thus, for any tree $t$ valid for $D$, there exists a CSS rule $r_i = L_{i1} \land L_{i2} \land L_{i3}$ that is applied to the leaf node $b$ in $t$. Furthermore, $\text{spec}(r_i) > \text{spec}(r_B)$. This implies that $r_B$ is never applied to the leaf node $b$, meaning that $r_B$ is unsatisfiable.

(⇐) Suppose that for any tree $t$ valid for $D$, $r_B$ is not applied to the leaf node $b$ in $t$. This implies that for any tree valid for $D$, at least one of $r_1, r_2, \ldots, r_m$ is applied to the leaf node in $t$. This implies that for any truth assignment to $\phi$, $\phi$ becomes true.

As shown above, the problem is also coNP-hard for CSS using only ‘$>$’ as a combinator.

**Theorem 2:** UNSAT($\text{CSS}^{\lt\lt}$) is coNP-hard under closure-free, duplicate-free, and non-recursive DTDs.

**Proof.** Similar to Theorem 1, this theorem can be proven through a reduction from the 3DNF-tautology problem. Let $\phi = (l_{11} \land l_{12} \land l_{13}) \lor (l_{21} \land l_{22} \land l_{23}) \lor \cdots \lor (l_{m1} \land l_{m2} \land l_{m3})$ be a 3DNF formula. The DTD $D = (d, s)$ is the same as the DTD in Theorem 1. Then, let $R = [r_1, r_2, \ldots, r_m, r_B]$ be a CSS code. Here, $r_B = b \land p : v$ and $r_i (1 \leq i \leq m)$ is defined so that ‘$\land$’ is simulated by consecutive ‘$*$’ selectors connected by ‘$>$’ combinators. That is, $\text{dist}(L_{i1}, L_{i2})$’s connected by ‘$>$’

\[
\begin{align*}
  r_i &= L_{i1} \land \cdots \land b \land p : v_i, \\
  r_B &= b \land p : v,
\end{align*}
\]

and $L_{ij}$ is the same as Eq. (1). Moreover, for $L_{ij} \in \{X_{iT}, X_{iF}\}$ and $L_{ij+1} \in \{X_{iT}, X_{iF}, b\}$, we define

\[
\begin{align*}
  \text{dist}(L_{ij}, L_{ij+1}) &= \begin{cases} 
  k' = k - 1, & \text{if } L_{ij+1} \in \{X_{iT}, X_{iF}\}, \\
  n - k, & \text{if } L_{ij+1} = 'b',
\end{cases}
\end{align*}
\]

where $n$ is the number of variables in $\phi$. For example, suppose that the $i$th term of $\phi$ is $(x_2 \lor \neg x_5 \lor x_7)$ and that $n = 9$. Then we have

\[
\begin{align*}
  r_i &= X_{iT} \land \cdots \land b \land p : v_i,
\end{align*}
\]

where (a) $\text{dist}(X_{iT}, X_{iF}) = 5 - 2 - 1 = 2$, (b) $\text{dist}(X_{iF}, X_{iT}) = 7 - 5 - 1 = 1$, and (c) $\text{dist}(X_{iT}, b) = 9 - 7 = 2$.

Similar to Theorem 1, we can show that $\phi$ is a tautology if and only if $r_B$ is unsatisfiable under $D$ w.r.t. $R$.

We next consider CSS rules using sibling combinators. Firstly, the problem is coNP-hard for CSS rules using only ‘$-$’ as a combinator.

**Theorem 3:** UNSAT($\text{CSS}^{\lt\lt}$) is coNP-hard under closure-free, duplicate-free, and non-recursive DTDs.

**Proof.** Similar to Theorem 1, this theorem can be proven through a reduction from the 3DNF-tautology problem. Let $\phi = (l_{11} \land l_{12} \land l_{13}) \lor (l_{21} \land l_{22} \land l_{23}) \lor \cdots \lor (l_{m1} \land l_{m2} \land l_{m3})$ be a 3DNF formula. From $\phi$, we define a DTD $D$, a CSS code $R$ in which each rule uses only ‘$-$’ as a combinator, and a CSS rule $r \in R$. First, the DTD $D = (d, s)$ is defined as follows:

\[
\begin{align*}
  d(s) &= (X_{iT} | X_{iF}) (X_{iT} | X_{iF}) \\
  \cdots (X_{iT} | X_{iF}) b, \\
  d(X_{iT}) &= d(X_{iF}) = \epsilon, \quad (1 \leq i \leq n) \\
  d(b) &= \epsilon.
\end{align*}
\]

Thus, for any tree $t$ valid for $D$, the children of $s$ represent a truth assignment to $\phi$.

Second, let $R = [r_1, r_2, \ldots, r_m, r_B]$ be a CSS code, where

\[
\begin{align*}
  r_i &= L_{i1} \land L_{i2} \land L_{i3} \land b \land p : v_i, \\
  r_B &= b \land p : v,
\end{align*}
\]

and $L_{ij}$ is the same as Eq. (1).

Similar to Theorem 1, we can show that $\phi$ is a tautology if and only if $r_B$ is unsatisfiable under $D$ w.r.t. $R$.

The problem is also coNP-hard for CSS rules using only ‘$+$’ as a combinator.

**Theorem 4:** UNSAT($\text{CSS}^{\lt\lt}$) is coNP-hard under closure-free, duplicate-free, and non-recursive DTDs.

**Proof.** This theorem can be proven in a manner similar to Theorem 3. Let $\phi = (l_{11} \land l_{12} \land l_{13}) \lor (l_{21} \land l_{22} \land l_{23}) \lor \cdots \lor (l_{m1} \land l_{m2} \land l_{m3})$ be a 3DNF formula. The DTD $D$ is the same as that in Theorem 3. Let $R = [r_1, r_2, \ldots, r_m, r_B]$ be a CSS code, where $r_B = b \land p : v$, and $r_i$ is a CSS rule of the following form:

\[
\begin{align*}
  \text{dist}(L_{i1}, L_{i2}) & \text{’s connected by ‘$+$’} \\
  \text{dist}(L_{i3}, L_{i4}) & \text{’s connected by ‘$+$’} \\
  r_i &= L_{i1} \lor \cdots \lor L_{i3} \lor b \land p : v_i, \\
  \text{dist}(L_{i3}, b) & \text{’s connected by ‘$+$’}
\end{align*}
\]

and $L_{ij}$ is the same as Eq. (1) and $\text{dist}(\cdot)$ is defined as Eq. (2).

Now, it is easy to show that $\phi$ is a tautology if and only if $r_B$ is unsatisfiable under $D$ w.r.t. $R$. 

3.1.1 Lower Bound under Disjunction-Free DTDs

For the XPath satisfiability problem, some tractable cases can be obtained by restricting DTDs to be disjunction-free [1]. We consider the lower bound of complexity for the CSS unsatisfiability problem under disjunction-free DTDs.

We first consider CSS using ‘*’ or ‘>’ as a combinator. We have the following theorem.

**Theorem 5:** \( \text{UNSAT}(\text{CSS}^{*}) \) and \( \text{UNSAT}(\text{CSS}^{>}) \) are coNP-hard under disjunction-free, closure-free, duplicate-free, and non-recursive DTDs.

**Proof.** First consider CSS using ‘*’. Let \( \phi = (l_{11} \land l_{12} \land l_{13}) \lor (l_{21} \land l_{22} \land l_{23}) \lor \cdots \lor (l_{m1} \land l_{m2} \land l_{m3}) \) be a 3DNF formula. From \( \phi \), CSS code \( R = [r_1, r_2, \ldots, r_B] \) is defined as same as the proof of Theorem 1. As for DTD, let \( D = (d, s) \) be the DTD defined in the proof of Theorem 1. We use DTD \( D’ = (d’, s) \), which is obtained from \( D \) by replacing each disjunction operator in \( d \) with a concatenation operator. Thus \( d’ \) is defined as follows.

\[
\begin{align*}
    d'(s) &= X_{1T} X_{1F}, \\
    d'(X_{1T}) &= d'(X_{1F}) = X_{2T} X_{2F}, \\
    d'(X_{2T}) &= d'(X_{2F}) = X_{3T} X_{3F}, \\
    & \vdots \\
    d'(X_{n-1T}) &= d'(X_{n-1F}) = X_{nT} X_{nF}, \\
    d'(X_{nT}) &= d'(X_{nF}) = b, \\
    d'(b) &= \epsilon.
\end{align*}
\]

This reduction can be done in polynomial time. Since \( D’ \) has no disjunction, any valid tree \( t \) of \( D’ \) has the same structure; the root is labeled by \( s \), which has two children labeled by \( X_{1T} \) and \( X_{1F} \), respectively, and so on. Figure 3 depicts a valid tree of \( D’ \) with \( n = 3 \). Therefore, each path in \( t \) from the root \( s \) to a leaf \( b \) corresponds to a valid tree of \( D \), and vice versa. This implies that \( \phi \) is a tautology if and only if \( r_B \) is unsatisfiable. The case for CSS using ‘>’ can be shown similar to above, by using \( D’ \) and the CSS code of the proof of Theorem 2.

Second, we consider CSS using sibling combinators. Under disjunction-free and closure-free DTDs, each content model represents a single string rather than a set of strings. This implies the following theorem:

**Theorem 6:** \( \text{UNSAT}(\text{CSS}^{*\lor}) \) is in PTIME under disjunction-free and closure-free DTDs.

**Sketch of Proof.** Let \( D = (d, s) \) be a disjunction-free and closure-free DTD. Then for any \( a \in \Sigma \), \( d(a) \) is a string. Then for any CSS rule \( r \in R \), we can identify the substring(s) \( \phi \) of \( d(a) \) matched by \( \text{sel}(r) \). Thus for a given CSS rule \( r \), by comparing (a) the substring(s) matched by \( \text{sel}(r) \) and (b) those matched by the selectors of the rules conflicting with \( r \), we can determine whether \( r \) is satisfiable.

However, the above tractability does not hold without closure-freeness.

**Theorem 7:** \( \text{UNSAT}(\text{CSS}^{*\lor}) \) is coNP-hard under disjunction-free and non-recursive DTDs.

**Proof.** Let \( \phi = (l_{11} \land l_{12} \land l_{13}) \lor (l_{21} \land l_{22} \land l_{23}) \lor \cdots \lor (l_{m1} \land l_{m2} \land l_{m3}) \) be a 3DNF formula and let \( x_1, x_2, \ldots, x_n \) be the variables in \( \phi \). We define the DTD \( D = (d, s) \) as follows:

\[
\begin{align*}
    d(s) &= (X_{1F_1} Y_{1})(X_{2F_2} Y_{2}) \cdots (X_{nF_n} Y_{n})b, \\
    d(X_i) &= d(F_i) = d(Y_i) = \epsilon. \quad (1 \leq i \leq n)
\end{align*}
\]

Here, \( X_{iF_1} Y_{i} \) indicates the value of \( x_i \). If the children of \( s \) are matched by \( X_{iF} Y_{i} \), without \( \phi \), then we consider \( x_i \) to be true. If the children of \( s \) contain one or more \( F_i \), then we consider \( x_i \) to be false. Let \( R = [r_1, r_2, \ldots, r_B] \) be a CSS code, where

\[
\begin{align*}
    r_i &= L_{i1} \sim L_{i2} \sim L_{i3} \sim b \ p : v_i, \quad (1 \leq i \leq m) \\
    r_B &= b \ p : v.
\end{align*}
\]

Here, \( L_{ij} \) is defined as follows \((1 \leq i \leq m, 1 \leq j \leq 3)\):

\[
L_{ij} = \begin{cases} 
    X_k Y_k & \text{if } l_{ij} = x_k, \\
    X_k F_k & \text{if } l_{ij} = \neg x_k.
\end{cases}
\]

Now it is easy to show that for any truth assignment, \( \phi \) is true if and only if \( r_B \) is unsatisfiable under \( D \) w.r.t. \( R \).

**Theorem 8:** \( \text{UNSAT}(\text{CSS}^{*\lor}) \) is coNP-hard under disjunction-free and non-recursive DTDs.

**Proof.** Let \( \phi = (l_{11} \land l_{12} \land l_{13}) \lor (l_{21} \land l_{22} \land l_{23}) \lor \cdots \lor (l_{m1} \land l_{m2} \land l_{m3}) \) be a 3DNF formula. We define the DTD \( D = (d, s) \) as follows:

\[
\begin{align*}
    d(s) &= (U_{1T_1} T_{1F_1} X_{1})(U_{2T_2} T_{2F_2} X_{2}) \cdots (U_{nT_n} T_{nF_n} X_{n})b \\
    d(U_i) &= d(T_i) = d(F_i) = d(X_i) = \epsilon. \quad (1 \leq i \leq n)
\end{align*}
\]

Here, \( U_i T_{iF} X_i \) in \( d(s) \) indicates the value of \( x_i \). If the children of \( s \) contain one \( T_i \), but no \( F_i \), then we consider \( x_i \) to be true. If the children of \( s \) contain one \( F_i \), but no \( T_i \), then we consider \( x_i \) to be false (all other cases are irrelevant). Let \( R = [r_1, r_2, \ldots, r_m, r_B] \) be a CSS code, where
Then, a selector \( \text{sel} \) matches an ext-path \( v_1, v_2, \ldots, v_n \) if and only if

\[
\begin{align*}
U_j + T_j + X_j & \quad \text{if } x_j \text{ appears in } C_i \text{ without negation}, \\
U_j + F_j + X_j & \quad \text{if } x_j \text{ appears in } C_i \text{ with negation}, \\
U_j + * + X_j & \quad \text{otherwise},
\end{align*}
\]

Let \( C_i \) denote the \( i \)-th clause of \( \phi \). Then \( L_{ij} \) and \( L'_j \) are defined as follows (\( 1 \leq j \leq n \)):

\[
L_{ij} = \begin{cases} 
U_j + T_j + X_j & \text{if } x_j \text{ appears in } C_i \text{ without negation}, \\
U_j + F_j + X_j & \text{if } x_j \text{ appears in } C_i \text{ with negation}, \\
U_j + * + X_j & \text{otherwise},
\end{cases}
\]

\[
L'_j = U_j + * + X_j.
\]

Now it is easy to show that \( \phi \) is true for any truth assignment if and only if \( r_g \) is unsatisfiable under \( D \), i.e., for any valid instance of \( D \) such that \( \text{sel}(r_g) \) matches the instance, there exists a rule \( r_i \in R \) such that \( \text{sel}(r_i) \) matches the instance. \( \square \)

### 3.2 Upper Bound

We next consider the upper bound of complexity for the problem. We first consider the case of restricted DTDs, then consider the case of general DTDs.

Let \( t \) be a tree. We use an “extended” path that allows horizontal traversal in addition to parent-child traversal. Formally, an extended path (ext-path for short) of \( t \) is a sequence \( v_1, v_2, \ldots, v_n \) of nodes such that \( v_1 \) is the root of \( t \) and that for every \( 1 \leq i \leq n - 1 \), \( v_{i+1} \) is either a child or the immediate right sibling of \( v_i \). In particular, we say that an ext-path \( \text{path} = v_1, v_2, \ldots, v_n \) is straight if \( v_{i+1} \) is a child of \( v_i \) for every \( 1 \leq i \leq n - 1 \). Then, a selector \( \text{sel} \) matches an ext-path \( v_1, v_2, \ldots, v_n \) if for some \( v_i \) such that \( v_{i+1} \) is a child of \( v_i \) for every \( 1 \leq j \leq i - 1 \), \( \text{sel} \) matches \( (v_i, v_{i+1}) \). For example, consider the tree \( t \) depicted in Fig. 4. Then \( p = v_a, v_c, v_d, v_e, v_f, v_g \) is an ext-path of \( t \). For a selector \( \text{sel} = d - f > g \), \( \text{sel} \) matches \( p \) since \( v_c \) is a child of \( v_a \), \( v_d \) is a child of \( v_c \), and \( \text{sel} \) matches \( (v_d, v_g) \).

#### 3.2.1 The Case of Restricted DTDs

**Theorem 9:** \( \text{UNSAT}(\text{CSS}^{\omega > 1}) \) is in \( \text{coNP} \) under non-recursive DTDs.

**Sketch of Proof.** We show that the following “satisfiability” problem is in \( \text{NP} \), which implies that \( \text{UNSAT}(\text{CSS}^{\omega > 1}) \) is in \( \text{coNP} \).

**Input:** a non-recursive DTD \( D = (d, s) \), a CSS code \( R \), and a CSS rule \( r \in R \)

**Problem:** determine if there exists a straight ext-path \( \text{path} = v_1, v_2, \ldots, v_n \) satisfying the following conditions:

1. \( \text{path} \) is “valid” for \( D \), that is, there exists a tree \( t \) valid for \( D \) containing \( \text{path} \).
2. \( \text{sel}(r) \) matches \( \text{path} \), but for any \( r' \in R \), \( \text{sel}(r') \) does not match \( \text{path} \) whenever \( \text{prop}(r') = \text{prop}(r) \) and (a) \( \text{spec}(\text{sel}(r')) \geq \text{spec}(\text{sel}(r)) \) or (b) \( \text{spec}(\text{sel}(r')) = \text{spec}(\text{sel}(r)) \) and \( \text{index}_R(r') > \text{index}_R(r) \).

Here, suppose that an “answer” \( \text{path} \) satisfying the above conditions (i) and (ii) is guessed. Since \( D \) is non-recursive, the length of \( \text{path} \) is bounded by the size of \( D \). Therefore, the conditions (i) and (ii) can be verified in polynomial time by using \( D \) and \( R \), meaning the above problem is in \( \text{NP} \). \( \square \)

#### 3.2.2 The Case of General DTDs

In the following, we consider DTDs without any restrictions. First, consider CSS rules using only uses only ‘\( \cdot \)’ as combinators. We have the following theorem.

**Theorem 10:** \( \text{UNSAT}(\text{CSS}^{\omega > 1}) \) is in \( \text{coNP} \) under general DTDs.

**Proof.** Let \( D = (d, s) \) be a DTD over \( \Sigma \), \( R \) be a CSS code, and \( r \in R \). Basically, this theorem can be shown similar to Theorem 9. However, under general DTDs the length of an “answer” ext-path is not bounded. But this can be addressed as follows. Let \( \text{spl}(a, D) \) be the length of the shortest straight valid ext-path from the root to a node labeled by \( a \). Then \( \text{spl}(a, D) \) is no more than \( |\Sigma| \) since the shortest valid path is “non-recursive.” Let \( \text{maxspl}(D) = \max_{a \in \Sigma} \text{spl}(a, D) \). Moreover, let \( \text{maxsel}(R) = \max_{a \in \Sigma} \text{len}(\text{sel}(r)) \). Then we have that, for a CSS rule \( r \) using only ‘\( \cdot \)’ as combinators, there is a straight valid ext-path to which \( r \) is applied if and only if there is a straight valid ext-path to which \( r \) is applied such that the length of \( \text{path} \) is no more than \( \text{maxspl}(D) + \text{maxsel}(R) \). To see this, let \( \text{path} = v_1, v_2, \ldots, v_n \) be a straight valid ext-path. To check if \( r \) is applied to \( v_n \) on \( \text{path} \), we need not to check all the nodes of \( \text{path} \); a suffix of \( \text{path} \) of at most length \( \text{maxsel}(R) \) suffices. Let \( v_j, v_{j+1}, \ldots, v_n \) be such a suffix of \( \text{path} \), and consider the prefix of \( \text{path} \) from \( v_1 \) to \( v_j \). The length of the prefix is not bounded, but the prefix can be replaced by a “shortest” straight valid ext-path \( v'_1, v'_2, \ldots, v'_k \) such that \( l(v'_i) = s \), \( l(v'_j) = l(v_j) \), and \( k \leq \text{maxspl}(D) \). The resulting ext-path \( \text{path}' \) is a valid ext-path of length no more than \( \text{maxspl}(D) + \text{maxsel}(R) \) such that \( r \) is applied to \( v_n \) on \( \text{path} \) if and only if \( r \) is applied to \( v_j \) on \( \text{path}' \). \( \square \)

We next consider the upper bound for
UNSAT(CSS($\omega^{a_{i}}$-$\omega^{\cdot}$.)). To show the upper bound, we define DTD automaton that represents the structure of DTD. For example, Fig. 5(a) illustrates the DTD automaton of $D = (d, s)$, where $d(s) = ab^*$, $d(a) = e$, $d(b) = sc$, and $d(c) = e$. As shown in the figure, the horizontal transitions represent sibling relationships between elements, and the vertical transitions (with subscript ‘$v$‘) represent parent-child relationships between elements.

To define the DTD automaton, we use Glushkov automaton (or position automaton)[8], [17]. Let $r$ be a regular expression. Each label occurring in $r$ is superscripted with a number to distinguish different occurrences of the same label in $r$. By $r^a$ we mean the superscripted regular expression of $r$ obtained by superscripting each label occurring in $r$. By $\text{sym}(r^a)$ we mean the set of superscripted labels occurring in $r^a$. For example, if $r = (a(b)(ab)^*b$, then $r^a = (a^1b^1)(a^2b^2)^*b^3$ and $\text{sym}(r^a) = \{a^1, b^1, a^2, b^2, b^3\}$. Let $d^a$ be a superscripted label of $a$. By $\langle d^a \rangle$ we mean the label resulting from $d^a$ by dropping the superscript of $d^a$, namely $\langle d^a \rangle = a$. For a word $w$ of superscripted labels, $w^a$ denotes the word obtained by dropping the superscript of each label in $w$. In the following, we use $u,x,y,z$ to denote superscripted labels.

The Glushkov automaton of $r$ is a five-tuple $G = (Q, \Sigma, \delta, q_0, F)$, where $Q = \text{sym}(r^a) \cup \{q_0\}$ is a set of states, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, $q_0 \notin \text{sym}(r^a)$ is the initial state of $G$, and $F$ is a set of final states. To define $\delta$ and $F$, we need $\text{First}(r^a)$, $\text{Last}(r^a)$, and $\text{Follow}(r^a, x)$, which are defined as follows:

First($r^a$) = \{ $x \in \text{sym}(r^a)$ \mid $xw \in L(r^a)$ for some word $w$, \}

Last($r^a$) = \{ $x \in \text{sym}(r^a)$ \mid $wx \in L(r^a)$ for some word $w$, \}

Follow($r^a$, $x$) = \{ $y \in \text{sym}(r^a)$ \mid $wxyu \in L(r^a)$ for some words $w$, $u$, $y$, \}

Then, $\delta$ and $F$ are defined as follows.

$\delta(x, a) = \left\{ \begin{array}{ll} \{ y \in \text{First}(r^a), y^a = a \} & \text{if } x = q_0, \\
\{ y \in \text{Follow}(r^a, x), y^a = a \} & \text{otherwise,} \end{array} \right.$

For any regular expression $r$, it holds that $L(r) = \{ w^a \mid w \in L(G) \}$. The Glushkov automaton $G$ can be constructed in polynomial time [4].

We define the DTD automaton formally. Let $D = (d, s)$ be a DTD over $\Sigma$. Let $\Sigma_v = \{ a_i \mid a_i \in \Sigma \}$ be the set of vertical labels. Let $G_a = (Q_a, \Sigma, \delta_a, q_0, F_a)$ be the Glushkov automaton of $d(a)$. Without loss of generality, we assume that $Q_a \cap Q_b = \emptyset$ whenever $a \neq b$, where $Q_b$ is the set of states of the Glushkov automaton of $d(b)$. Then the DTD automaton of $D$ w.r.t. CSS rule $r$ is defined as an NFA $M = (Q, \Sigma_v \cup \Sigma, \delta, r^0, F)$, where $Q$, $\delta$, and $F$ are defined as follows:

- First, $Q$ is defined as follows:

  $Q = \bigcup_{a \in \Sigma} Q_a \cup \{ r^0, s^0 \},$

  where $Q_a$ is the set of states of $G_a$, $r^0$ is the initial state and $s^0$ is the state representing the start label $s$.

- $\delta$ is obtained by taking the union of (a) $\delta_a$ (the transition function of $G_a$) of each $a$ and (b) $\delta_v$, where $\delta_v$ represents “horizontal transitions” and $\delta_v$ represents “vertical transitions.” Here, $\delta_v$ is defined as follows:

  $\delta_v(x, c_{i}) = \left\{ \begin{array}{ll}
\{ y \in Q_v \mid y^v = c_{i} \} & \text{if } x = r^0 \text{ and } c_{i} = s_v, \\
\{ y \in Q_v \mid y^v = c_{i} \} & \text{if } x = s^0, \\
\emptyset & \text{otherwise.} \\
\end{array} \right.$

Now, $\delta : Q \times (\Sigma_v \cup \Sigma) \rightarrow Q$ is defined by using $\delta_a$ and $\delta_v$ as follows:

$\delta(x, c_{i}) = \left\{ \begin{array}{ll}
\delta_a(x, c_{i}) & \text{if } c_{i} \in \Sigma \text{ and } x \in Q_a \text{ for some } a, \\
\delta_v(x, c_{i}) & \text{if } c_{i} \in \Sigma_v. \\
\end{array} \right.$

- $F$ is determined by the last simple selector of $r$ as follows:

  $F = \{ x \mid x \in Q, x^a = \text{last}(\text{sel}(r)) \}.$

For example, let us consider Fig. 5. If $r = a_n c p : v$, then $F = \{ c^1 \}$.

One can see that the DTD automaton exactly covers every valid ext-path.

**Lemma 1:** Let $D = (d, s)$ be a DTD and $M = (Q, \Sigma_v \cup \Sigma, \delta, r^0, F)$ be the DTD automaton of $D$. Then, there exists a tree $t$ valid for $D$ containing an ext-path $p = v_1, v_2, \ldots, v_n$ if and only if there is a sequence $s = x_0, x_1, \ldots, x_n$ of states of $M$ corresponding to $p$. In other words, $p$ and $s$ satisfy

- $x_0 = r^0$ and $x_1 = s^0$, and
- for $2 \leq i \leq n$,
  - $x_i \in \delta(x_{i-1}, l(v_i))$ if $v_i$ is the immediate right sibling
  - $x_i \notin \delta(x_{i-1}, l(v_i))$ if $v_i$ is the immediate left sibling


Sketch of Proof. Let $sel = s_1c_1s_2 \cdots c_{n-1}s_n$ be a selector, where $s_i$ is a simple selector and $c_i$ is a combinator. We first define the regular expression representation of $sel$, denoted $re(sel)$, as follows: $re(sel) = c_0^{r_1}c_1^{r_2} \cdots c_{n-1}^{r_n}s_n$, where

$$c_i^r = \begin{cases} \Sigma_i & \text{if } i = 0 \text{ or } (b) i \geq 1 \text{ and } c_i = ‘*’, \\ \epsilon & \text{if } c_i \in \{>, +\}, \\ \end{cases}$$

for $0 \leq i \leq n - 1$ and

$$s_i^r = \begin{cases} s_i & \text{if } s_i \in \Sigma, \text{ and } i = 1 \text{ or } c_{i-1} \in \{>, \}, \\ r_i & \text{if } s_i = ‘*’ \text{ and } c_{i-1} \in \{>, \}, \\ s_i & \text{if } s_i \in \Sigma \text{ and } c_{i-1} \in \{+, -\}, \\ l_{\leq} & \text{if } s_i = ‘*’ \text{ and } c_{i-1} \in \{+, -\}, \\ \end{cases}$$

for $1 \leq i \leq n$. For example, if $sel = a \cdot b \cdot c$, then $re(sel) = (\Sigma_0)^r a_1(\Sigma_1)b c_n$.

By using the DTD automaton and above regular expression representation of a selector, we can check if $r \in R$ is unsatisfiable under a DTD w.r.t. $R$ as follows:

1. Construct the DTD automaton $M$ of $D$.
2. Convert $sel(r)$ into its regular expression representation $re(sel(r))$.
3. Let $r_1, r_2, \ldots, r_k \in R$ be the CSS rules that may conflict with $r$. That is, for every $1 \leq i \leq k$, $r_i$ satisfies that $last(sel(r_i)) = last(sel(r))$, $prop(r_i) = prop(r)$, and (a) $spec(sel(r_i)) > spec(sel(r))$ or (b) $spec(sel(r_i)) = spec(sel(r))$ and $index_R(r_i) > index_R(r)$. Convert $sel(r_i)$ into its regular expression representation $re(sel(r_i))$ for every $1 \leq i \leq k$.
4. Determine if

$$L(are(re(sel(r_i)))) \cap L(M) \subseteq \bigcup_{1 \leq i \leq k} L(re(sel(r_i))) \cap L(M).$$

By this and Lemma 1, one can see that step 4 is true if and only if $r$ is unsatisfiable under $D$ w.r.t. $R$.

Since the containment problem for regular expression is PSPACE-complete [18], step 4 can be done in PSPACE. \qed

### 4.1 Conditions R1 and R2: Disabling Universal Selectors

In Theorem 2, we showed that $\text{UNSAT}(\text{CSS}^{(\ast)})$ is coNP-hard under very restricted DTDs. Recall that the proof of the coNP-hardness essentially depends on universal selector ‘*‘. In the following, we show that the problem becomes tractable even under general DTDs, if universal selector is disabled.

Let $R$ be a CSS code and $r \in R$. By $C_R(r)$ we mean the set of CSS rules in $R$ that may conflict with $r$, that is,

$$C_R(r) = \{ r' \in R \mid \text{last}(sel(r)) = \text{last}(sel(r')), \text{prop}(r) = \text{prop}(r'), \text{spec}(sel(r'))) > \text{spec}(sel(r)) \text{ or } \text{spec}(sel(r')) = \text{spec}(sel(r)) \text{ and } \text{index}_R(r') > \text{index}_R(r))\}.$$ 

If a selector $sel$ uses only child combiners and labels, then $sel$ is called a child-label selector. For example, $a > b > c$ is a child-label selector but $a > * > c$ is not. We show that if the following condition R1 holds, then whether a rule $r \in R$ is satisfiable under $D$ w.r.t. $R$ is determined in PTIME.

**R1:** $sel(r)$ is a child-label selector and for all $r' \in C_R(r)$, $sel(r')$ is a child-label selector.

Let $D = (d, s)$ be a DTD and $M = (\Sigma, \Sigma_\delta, r, F, \text{D})$ be the DTD automaton of $D$. For a child-label selector $sel = s_1 > s_2 > \cdots > s_n$ by $Q_l(sel)$ we mean the set of “start” states $q_0$ of a vertical path $q_0 \rightarrow q_1 \rightarrow \cdots \rightarrow q_n$ in $M$, that is,

$$Q_l(sel) = \{ q_0 \mid q_i \in \delta(q_{i-1}, (s_i)c_i) \text{ for } 1 \leq i \leq n, q_i \in Q \text{ for } 0 \leq i \leq n \}.$$ 

We now have the following theorem.

**Theorem 12:** If condition R1 holds for all $r \in R$, then $\text{UNSAT}(\text{CSS}^{(\ast)})$ is in PTIME under general DTDs.

**Sketch of Proof.** Let sel, sel' be child-label selectors, where $sel = s_1 > s_2 > \cdots > s_n$ and that $sel' = s'_1 > s'_2 > \cdots > s'_m$. We say that $sel$ is a suffix of $sel'$ if $n < m$ and $s_i = s'_{i+n-m}$ for every $1 \leq i \leq n$. By Condition R1, it suffices to consider only rules $r' \in C_R(r)$ such that $sel(r)$ is a suffix of $sel(r')$. Let

$$C_f^{ref}(r) = \{ r' \in C_R(r) \mid sel(r) \text{ is a suffix of } sel(r') \}.$$ 

Then it is easy to show that $r$ is satisfiable under $D$ w.r.t. $R$ if and only if there is a state $q \in Q$ such that

1. $q$ is reachable from $r_0$,
2. for some $q' \in Q_l(sel(r))$, $q'$ is reachable from $q$ over $M$, and that
3. for any $r' \in C_f^{ref}(r)$ and any $q' \in Q_l(sel(r'))$, $q'$ is not reachable from $q$ over $M$.

The second condition means that for some valid tree $t$, $r$ can be applied to a node on $t$. The third condition means that no
rule in $C^s_{R}(r)$ can be applied to the node. Both conditions can be checked in polynomial time. □

A similar argument can also be applied to $CSS^s$. By Theorem 4, $UNSAT(CSS^s)$ is coNP-hard under very restricted DTDs. However, similar to the above theorem, we can show that the problem can be solved in PTIME if universal selector is disabled.

We say that $sel$ is called a *is-label selector* if $sel$ uses only immediate sibling combinators and labels. We present the “horizontal” version of condition R1, as follows.

**R2:** $sel(r)$ is an is-label selector and for all $r' \in C_{R}(r)$, $sel(r')$ is an is-label selector.

Similar to Theorem 12, we can show the following theorem. The only difference to Theorem 12 is that this theorem uses horizontal transitions instead of vertical transitions.

**Theorem 13:** If condition R2 holds for all $r \in R$, then $UNSAT(CSS^s)$ is in PTIME under general DTDs. □

4.2 Condition R3: Restricting the Number of Conflicting Rules

The third condition is to restrict the number of conflicting rules and the form of their selectors.

To define the condition, we present a restricted form of selector. Consider dividing a selector $sel$ by its descendant/general sibling combinators. Then $sel$ can be denoted

$$sel = sel_1 sel_2 \cdots sel_m,$$

where $sel_i$ is a selector such that the first combinator is either a descendant or general sibling combinator and the others are child or immediate sibling combinators. We say that $sel_i$ is specific if each simple selector in $sel_i$ is a label and $s \neq s'$ for any distinct simple selectors $s, s'$ in $sel_i$. For example, if $sel_i = \langle a \text{ } b \text{ } c \rangle$, then $sel_i$ is specific but $sel_j$ is not. We say that $sel$ is specific if $sel_i$ is specific for every $1 \leq i \leq m$.

Let $c > 0$ be a constant number. We show that if the following condition holds, then the unsatisfiability of $r$ can be determined in polynomial time.

**R3:** (1) $|C_{R}(r)| < c$ and (2) if $|C_{R}(r)| \geq 1$, then $sel(r')$ is specific for any $r' \in C_{R}(r)$.

**Theorem 14:** If condition R3 holds for all $r \in R$, then $UNSAT(CSS^{\omega,+,\cdot})$ is in PTIME under general DTDs.

*Sketch of Proof.* Let $r \in R$ be a CSS rule. If $|C_{R}(r)| = 0$, then whether $r$ is unsatisfiable can be determined in PTIME by taking the intersection of the DTD automaton of $D$ and the NFA of $re(sel(r))$. Consider the case where $|C_{R}(r)| \geq 1$. Assume that $|C_{R}(r)| < c$ and that $sel(r')$ is specific for any $r' \in C_{R}(r)$. Then $sel(r')$ can be represented by a regular expression $re(sel(r'))$, which is represented by an NFA. But in this case $sel(r')$ is very restricted, i.e., specific. This implies that we can construct a DFA $M(sel(r'))$ equivalent to $re(sel(r'))$ without exponential state explosion (a more detailed construction is presented in the appendix).

By using $M(sel(r'))$, whether $r$ is unsatisfiable w.r.t. $R$ under $D$ can be determined as follows:

1. Construct $M(sel(r))$ from $sel(r)$.
2. Let $C_{R}(r) = \{r'_1, r'_2, \ldots, r'_m\} (m < c)$. Construct a DFA $M(sel(r'_i))$ for $1 \leq i \leq m$.
3. Construct the following NFA:

$$M'_r = M(sel(r)) \cap \neg M(sel(r'_1)) \cap \neg M(sel(r'_2)) \cap \cdots \cap \neg M(sel(r'_m)).$$

4. Construct intersection $M_r = M'_r \cap M_D$, where $M_D$ is the DTD automaton of $D$.

5. Determine if $L(M_r)$ is empty. Then it is easy to see that $M_r$ is empty if and only if $r$ is unsatisfiable w.r.t. $R$ under $D$.

Since $M(sel(r'_i))$ is deterministic, $\neg M(sel(r'_i))$ can be obtained in linear time. Furthermore, $M'_r$ is the intersection of $m$ DFAs and one NFA, which can be obtained in polynomial time since $m < c$. Therefore, $M'_r$ can be obtained in polynomial time. □

4.3 Condition R4: Restricting Length of CSS Rules

We now present the last condition under which the problem can be solved in PTIME. In short, the problem becomes tractable by restricting the length of selectors to no greater than two, even if there is no restriction on the number of conflicting rules and their selectors.

For a CSS rule $r$ and a DTD $D$, to check if $r$ is unsatisfiable, we use the DTD automaton $M$ of $D$ and find states that are matched by $r$ but not matched by the rules conflicting with $r$. In general, such states cannot be found efficiently according to the results presented in the previous section. However, we show that such a path can be found efficiently under the following condition:

**R4:** If $|C_{R}(r)| \geq 1$, then (1) $len(sel(r)) \leq 2$ and $len(re(sel(r))) = 2$ for every $r' \in C_{R}(r)$, and (2) the simple selectors of $r$ and any rule in $C_{R}(r)$ are labels.

To present an algorithm for solving the problem under condition R4, we give some definitions. Assuming condition R4 with $|C_{R}(r)| \geq 1$, by $C_{R}^c(r)$ we mean the set of CSS rules in $C_{R}(r)$ whose combinator is $c$, that is,

$$C_{R}^c(r) = \{r' \in C_{R}(r) \mid cmb(sel(r')) = c\},$$

where $c \in \{\omega, +, \cdot\}$ and $cmb(sel(r'))$ denotes the combinator of $sel(r')$ (since $len(sel(r')) = 2$, $sel(r')$ has exactly one combinator). We also use the following abbreviation:

$$first(C_{R}^c(r)) = \{first(sel(r')) \mid r' \in C_{R}^c(r)\}.$$

For example, let $R = [r_1, r_2, r_3, r_4]$, where

$$r_1 = a_{-}b_p : v_1,$$

$$r_2 = b_p : v_2,$$
Then \( C_R(r_1) = \{r_3, r_4\} \) and \( \text{first}(C_R(r_1)) = \{c, d\} \).

Under condition R4, we can focus on only ext-paths with a simple shape. Consider the case where \( r \) is a CSS rule with \( \text{cmb}(sel(r)) = \cdot \cdot \cdot \). Then any ext-path we need to check is a straight ext-path \( p \) together with siblings of the “bottom” node \( v_n = v_i' \) (Fig. 6). To see this, suppose that \( sel(r) \) matches \( p \). To check if \( r \) is applied to \( v_p \), we need to verify that none of the rules in \( C_R(r) \) is applied to \( v_n \), where \( C_R(r) = C_R(r) \cup C^e_R(r) \cup C^c_R(r) \cup C^s_R(r) \). For rules in \( C^e_R(r) \cup C^c_R(r) \), this can be checked by traversing only \( p \). For rules in \( C^s_R(r) \cup C^c_R(r) \), this can be done by checking only the labels of (left) siblings of \( v_n = v_i' \) - siblings of \( v_n \) are matched by \( C_R(r) \). Similar arguments can also be applied to the other cases. Thus, to determine if \( r \) is satisfiable under a DTD \( D \), our algorithm explores the DTD automaton of \( D \) and checks if \( D \) allows such a straight ext-path (with the siblings of the bottom node) to which \( r \) can be applied.

Let \( M = (Q, \Sigma_q, \Sigma, \delta, r^0, F) \) be the DTD automaton of \( D \). Our algorithm firstly explores \( M \) and finds states that are matched by \( r \) but not matched by the rules in \( C_R(r) \). To obtain such states, for a set \( S \subseteq Q \) and a set \( Skip \) of labels, we define the set of pairs of parent-child states \((x, y)\) such that \( y \) is vertically reachable in “i-hops” from a state in \( S \) without using \( Skip \), denoted \( RS^i(x, S, Skip) \), as follows:

\[
RS^i(x, S, Skip) = \begin{cases} 
\{(x, y) \mid x \in S, x^i \notin Skip, \ y \in \delta(x, c_i), c_i \in \Sigma\} & \text{if } i = 1, \\
\{(x, y) \mid x \in RS^{i-1}_c(M, S, Skip), \ x^i \notin Skip, y \in \delta(x, c_i), c_i \in \Sigma\} & \text{if } i \geq 2.
\end{cases}
\]

Then the “any-hops” version of \( RS^i(x, S, Skip) \), denoted \( RS^\infty(x, S, Skip) \), is defined as follows:

\[
RS^\infty(x, S, Skip) = \bigcup_{i \geq 1} RS^i(x, S, Skip).
\]

By \( RS^\infty(x, S, Skip, a) \) we mean the set of pairs of parent-child states \((x, y)\) in \( RS^\infty(x, S, Skip) \) such that \( y^\infty = a \), that is,

\[
RS^\infty(x, S, Skip, a) = \{(x, y) \mid (x, y) \in RS^\infty(x, S, Skip), y^\infty = a\}.
\]

### Algorithm 1 Check Unsatisfiability

**Input:** DTD \( D = (d, s) \), CSS code \( R \) and CSS rule \( r \in R \) satisfying condition R4  

**Output:** “satisfiable” or “unsatisfiable”

1: Construct the DTD automaton \( M = (Q, \Sigma_\Sigma, \delta, r^0, F) \) of \( D \).
2: \( N_1 \leftarrow RS^\infty(M, \{r^0\}, \text{first}(C_R(r)), \text{first}(sel(r))). \)
3: if \( \text{cmb}(sel(r)) = \cdot \cdot \cdot \) then
4: \( \text{if } \text{cmb}(sel(r)) = \cdot \cdot \cdot \) then
5: \( N_2 \leftarrow RS^\infty(M, \pi_2(N_1), \text{first}(C_R(r)), \text{last}(sel(r))). \)
6: else
7: \( N_2 \leftarrow RS^\infty(M, \pi_2(N_1), \text{first}(C_R(r)), \text{last}(sel(r))). \)
8: if for some \((y_i, z) \in N_2, y^\infty \notin \text{first}(C_R(r)) \) then there is a sequence \( q^0, q^1, \ldots, q^n \) of states of \( G_R \) satisfying the following:
9: \( q^0 \) is the initial state, \( q^i = z \) for some \( 1 \leq i \leq m, q^n \) is a final state, and \( q' \in \delta(q^{i-1}, (q^i)^\infty) \) for every \( 1 \leq i \leq m \).
10: \( q^j \notin \text{first}(C_R(r)) \) for every \( 1 \leq i \leq k \), and
11: else if \( \text{cmb}(sel(r)) = \cdot \cdot \cdot \) then
12: if for some \((x, y) \in N_1, x^i \notin \text{first}(C_R(r)) \) and there is a sequence \( q^0, q^1, \ldots, q^n \) of states of \( G_R \) satisfying the following:
13: \( q^0 \) is the initial state, \( q^i = y \) for some \( 1 \leq i \leq m, q^n \) is a final state, and \( q' \in \delta(q^{i-1}, (q^i)^\infty) \) for every \( 1 \leq i \leq m \).
14: \( q^{j-1} \notin \text{first}(C_R(r)) \) for every \( 1 \leq i \leq k-1 \), and
15: \( q^{k-1} \notin \text{first}(C_R(r)) \).
16: \( \text{return “satisfiable”}. \)
17: \( \text{return “unsatisfiable”}. \)
18: \( \text{return “unsatisfiable”}. \)

We now present our algorithm (Algorithm 1). For simplicity, in the following we present an algorithm for the case of \( \text{len}(sel(r)) = 2 \) (the algorithm for the case of \( \text{len}(sel(r)) = 1 \) can be obtained similarly). Under condition R4, Algorithm 1 checks if a given CSS rule \( r \in R \) is unsatisfiable under \( D \). The algorithm has two parts: the first part (lines 3 to 10) checks the satisfiability of \( r \) in the case of \( \text{cmb}(sel(r)) = \cdot \cdot \cdot \), and the second part (lines 11 to 14) checks the satisfiability of \( r \) in the case of \( \text{cmb}(sel(r)) = \cdot \cdot \cdot \). The outline of the first part is shown in Fig. 7. This part finds a state \( z \) that is reachable from \( r^0 \) via \( sel(r) \) and is not “blocked” by any rule in \( C_R(r) \) (lines 2 to 7), and then checks if \( z \) is not “blocked” by any rule in \( C_R(r), C^s_R(r), C^c_R(r) \) (line 8). In line 2, \( N_1 \) is the set of pairs of states \((x, y)\) such that \( y^\infty = \text{first}(sel(r)) \) and that \( y \) is vertically reachable from \( r^0 \) without using any label in \( \text{first}(C_R(r)) \). In lines 5 and 7, by \( \pi_2(N_1) \) we mean the second column of \( N_1 \), i.e., \( \pi_2(N_1) = \{y \mid (x, y) \in N_1\} \). Thus \( N_2 \) is the set of parent-child pairs \((y^j, z)\) such that \( z^\infty = \text{last}(sel(r)) \) and that \( z \) is vertically reachable in any-hops (line 5) or one-hop (line 7) from some state \( y \in \pi_2(N_1) \) without using any label in \( \text{first}(C_R(r)) \). Finally, line 8 checks if \( z \) is not blocked by any rule in \( C^s_R(r), C^c_R(r), \) and \( C^s_R(r) \). In line 8, \( G_{\cdot \cdot \cdot} \) denotes the Glushkov aut-
Lemma 2: Let $D = (d, s)$ be a DTD and $M = (Q, \Sigma \cup \Sigma, \Delta, r^0, F)$ be the DTD automaton of $D$. Then, there exists a tree $t$ valid for $D$ containing a straight ext-path $p = v_1, v_2, \ldots, v_n$ with the siblings $v'_1, v'_2, \ldots, v'_m$ of $v_n$ (i.e., $v_n = v'_j$ for some $j$) if and only if there is a sequence $x_0, x_1, \ldots, x_n$ of states of $M$ satisfying the following:

- $x_0 = r^0$,
- $x_i \in \delta(x_{i-1}, l(v_i))$ for every $1 \leq i \leq n$, and
- there is a sequence $q^0, q^1, \ldots, q^m$ of states corresponding to the siblings of $v_n$, i.e.,
  
  - $q^0$ is the initial state of $G_{(x_n)^{\bar{z}}}$, $q^k = x_n$ for some $k$, and $q^m$ is an accepting state of $G_{(x_n)^{\bar{z}}}$, and
  
  - $q^i \in \delta(q^{i-1}, l(v'_i))$ for every $1 \leq i \leq m$,

where $G_{(x_n)^{\bar{z}}}$ is the Glushkov automaton of $(x_n)^{\bar{z}}$.

We have the following theorem:

Theorem 15: Let $D = (d, s)$ be a DTD, $R$ be a CSS code, and $r \in R$. If condition R4 holds, then $r$ is unsatisfiable under $D$ w.r.t. $R$ if and only if $\text{CheckUnsatisfiability}$ returns "unsatisfiable".

Sketch of Proof. Assume that condition R4 holds. We show the case where $\text{cmb}(\text{sel}(r)) = \text{'\text{\text{'}}}$ (the other cases can be shown similarly). By condition R4, we have $C_G(r) = C_{G'}(r) \cup C_{G''}(r) \cup C_{G'''}(r) \cup C_{G''''}(r)$. Let $r$ be a tree valid for $D$. Consider a straight ext-path $p = v_1, v_2, \ldots, v_n$ in $t$ with the siblings $v'_1, v'_2, \ldots, v'_m$ of $v_n$ for some $k$ (Fig. 6). Here, suppose that $\text{sel}(r)$ matches $p$ but $r$ is not applied to $v_n$. By condition R4, the reason must be one of the following.

1. Some $r' \in C_G(r)$ is applied to $v_n$. In this case, it must hold that $l(v_i) = \text{first}(\text{sel}(r'))$ for some $1 \leq i \leq n - 1$ and that $l(v_n) = \text{last}(\text{sel}(r'))$.
2. Some $r' \in C_{G'}(r)$ is applied to $v_n$. In this case, it must hold that $l(v_{n-1}) = \text{first}(\text{sel}(r'))$ and that $l(v_n) = \text{last}(\text{sel}(r'))$.
3. Some $r' \in C_{G''}(r)$ is applied to $v_n$. In this case, $l(v_n) = \text{last}(\text{sel}(r'))$ and there must be a left sibling $v'_i (1 \leq i \leq k - 1)$ of $v_n = v'_k$ such that $l(v'_i) = \text{first}(\text{sel}(r'))$.
4. Some $r' \in C_{G'''}(r)$ is applied to $v_n$. In this case, $l(v_n) = \text{last}(\text{sel}(r'))$ and the immediate left sibling $v'_{k-1}$ of $v_n = v'_k$ must satisfy that $l(v'_{k-1}) = \text{first}(\text{sel}(r'))$.

Therefore, $r$ is satisfiable under $D$ w.r.t. $R$ if and only if there is a tree $t$ valid for $D$ containing a straight ext-path $p = v_1, v_2, \ldots, v_n$ with the siblings $v'_1, v'_2, \ldots, v'_m$ of $v_n$ such that $\text{sel}(r)$ matches $p$ and that none of the above conditions 1 to 4 holds.

In the following, we show that $r$ is unsatisfiable under $D$ w.r.t. $R$ if and only if $\text{CheckUnsatisfiability}$ returns "unsatisfiable".

$\Leftarrow\Rightarrow$ Suppose that $\text{CheckUnsatisfiability}$ returns "unsatisfiable" on line 10. Then (1) $N_2 = \emptyset$ or (2) for any $(y', z) \in N_2$, $y'^\sigma \in \text{first}(C_{G'''}(r))$ or there is no sequence $q^0, q^1, \ldots, q^m$ of states satisfying the conditions (a) to (c) on line 8. In the case of (1), any state matched by $\text{sel}(r)$ is also matched by $\text{sel}(r')$ for some $r' \in C_{G'''}(r)$. In the case of (2), $q^k = z$ is matched by $\text{sel}(r)$ but also matched by $\text{sel}(r')$ for some $r' \in C_{G'''}(r)$, $C_{G''}(r)$, or $C_{G'''}(r)$. In either case, by Lemma 2 there is no tree $t$ valid for $D$ containing a straight ext-path, say $p = v_1, v_2, \ldots, v_n$ such that $\text{sel}(r)$ matches $p$ but no rule in $C_{G'''}(r)$ is applied to $v_n$. Therefore, $r$ is unsatisfiable.

$\Rightarrow\Leftarrow$ Suppose that $\text{CheckUnsatisfiability}$ returns "satisfiable". Then there is a sequence $x_0, x_1, \ldots, x_n$ of states of $M$ satisfying the following condition:

1. From the construction of $N_1$ and $N_2$, we have $x_0 = r^0$, $x_i \in \delta(x_{i-1}, l(v_i))$ for every $1 \leq i \leq n$, $(x_i)^{\bar{z}} = \text{first}(\text{sel}(r))$ for some $1 \leq i < n$, $(x_n)^{\bar{z}} = \text{last}(\text{sel}(r))$, and $(x_i)^{\bar{z}} \notin \text{first}(C_{G'''}(r))$ for any $1 \leq i < n$.
2. From line 8, there is a sequence of states $q^0, q^1, \ldots, q^m$ such that $q^k = x_n$ for some $k$ and that the conditions (a) to (c) in line 8 holds.

By Lemma 2, there exists a tree $t$ valid for $D$ containing a
straight ext-path \( p = v_1, v_2, \ldots, v_n \) such that \( l(v_i) = (x_i) \) for every \( 1 \leq i \leq n \) and that \( v_n = v'_n \) has siblings \( v'_1, v'_2, \ldots, v'_m \) such that \( l(v'_i) = (q'_i) \) for every \( 1 \leq i \leq m \). By the conditions (a) to (c) in line 8, no rule in in \( C^\mu_P(r), C^\mu_S(r) \), or \( C^\mu_r(r) \) is applied to \( v_n \). Hence \( r \) is applied to \( v_n \) and thus \( r \) is satisfiable. \( \square \)

Consider the running time of the algorithm. \( R^\Sigma_S(M, S, \text{Skip}, a) \) can be obtained in polynomial time. The conditions in lines 8 and 12 can be checked by solving reachability problems with a few additional conditions. Hence it is easy to show that \text{CHECKUNSATISFIABILITY} \ runs \ in \ polynomial \ time. Therefore, we have the following:

**Theorem 16:** If condition R4 holds for all \( r \in R \), then \text{UNSAT}(\text{CSS}^{\rightarrow \cdots \rightarrow r}) \ is \ in \ \text{PTIME} \ under \ general \ DTDs.

5. Conclusion

In this paper, we considered the CSS unsatisfiability problem. First, we showed that the problem is coNP-hard under closure-free, duplicate-free, and non-recursive DTDs, even if only one of the four combinators of CSS is allowed. Next, we showed that the problem is coNP-hard even if DTDs are restricted to be disjunction-free and either child or descendant combinator is allowed. We also showed that the problem is in coNP or PSPACE depending on restrictions on DTDs and CSS. Finally, we presented conditions R1 to R4 under which the problem can be solved in polynomial time.

However, we still have much work to do. First, as shown in Table 1, the upper and lower bounds of complexity are not strict in several cases. Therefore, we need to identify the strict upper and lower bounds for the cases. In particular, we will investigate whether or not PSPACE-completeness of the problem holds when no restrictions are placed on DTDs or CSS. At present, we consider that the tight upper bound would be \( \Sigma^P \) or \( \Pi^P \) and we would like to try to prove it.

Second, we presented several conditions under which the CSS unsatisfiability problem is tractable, but it is not clear what extent these conditions are supported by real-world CSS codes. Therefore, we need to investigate real-world CSS codes to clarify this point. We expect that real-world CSS rules tend to have “short” selectors, meaning condition R4 could be supported by many of real-world CSS rules.

Third, we need to consider CSS selectors that were not considered in this paper, e.g., attributes (e.g., id and class), selectors using “first-child” and “last-child” pseudo classes, and so on.

Finally, we showed that the problem becomes tractable by restricting the length of selectors to no greater than two (condition R4). On the other hand, by Theorems 1 and 3, the problem becomes intractable in the case where the length of selectors is four. However, the (in)tractability of the case where the length of selectors is three is not identified, which is left as a future work.

**Acknowledgments**

The authors are thankful to anonymous reviewers for their insightful comments and suggestions.

**References**

[1] M. Benedikt, W. Fan, and F. Geerts, “XPath satisfiability in the presence of DTDs,” J. ACM, vol.55, no.2, pp.8.1–8.79, May 2008.

[2] B. Bos, D. Carlisle, P.D.F. Ion, B.R. Miller, and eds., “A MathML for CSS profile,” https://www.w3.org/TR/mathml-for-css/.

[3] M. Bosch, P. Geneves, and N. Layaida, “Automated refactoring for size reduction of CSS style sheets,” Proc. 2014 ACM Symposium on Document Engineering, DocEng ’14, pp.13–16, 2014.

[4] A. Brüggenmann-Klein, “Regular expressions into finite automata,” Theoretical Computer Science, vol.120, no.2, pp.197–213, 1993.

[5] Firebug Working Group, “FireBug,” https://www.getfirebug.com/.

[6] M.R. Garey and D.S. Johnson, Computers and Intractability - A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.

[7] P. Geneves, N. Layaida, and V. Quir, “On the analysis of cascading style sheets,” Proc. 21st International Conference on World Wide Web, WWW ’12, pp.809–818, 2012.

[8] V.M. Glushkov, “The abstract theory of automata,” Russian Math. Surveys, vol.16, pp.1–53, 1961.

[9] Google Inc., “Chrome developer tools,” https://developers.google.com/web/tools/chrome-devtools/.

[10] M. Hague, A.W. Lin, and C.-H.L. Ong, “Detecting redundant CSS rules in HTML5 applications: A tree rewriting approach,” SIGPLAN Not., vol.50, no.10, pp.1–19, Oct. 2015.

[11] O. Hartig, Querying a Web of Linked Data, IOS Press, 2016.

[12] Y. Ishihara, N. Suzuki, K. Hashimoto, S. Shimizu, and T. Fujiwara, “XPath satisfiability with parent axes or qualifiers is tractable under many of real-world DTDs,” Proc. 14th International Symposium on Database Programming Languages (DBPL 2013), Aug. 30, 2013, Riva del Garda, Trento, Italy, 2013.

[13] D. Mazinian and N. Tsantalis, “Migrating cascading style sheets to preprocessors by introducing mixins,” Proc. 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pp.672–683, 2016.

[14] D. Mazinian, N. Tsantalis, and A. Mesbah, “Discovering refactoring opportunities in cascading style sheets,” Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp.496–506, 2014.

[15] A. Mesbah and S. Mirshokraie, “Automated analysis of css rules to support style maintenance,” Proc. 34th International Conference on Software Engineering, ICSE ’12, pp.408–418, 2012.

[16] M. Montazerian, P.T. Wood, and S.R. Mousavi, “Xpath query satisfiability is in ptime for real-world DTDs,” Proc. 5th International Conference on Database and XML Technologies, XSym’07, pp.17–30, 2007.

[17] H. Yamada and R. McNaughton, “Regular expressions and state graphs for automata,” Ira Trans. Electron. Comput., vol.9, no.1, pp.39–47, 1960.

[18] I.J. Stockmeyer and A.R. Meyer, “Word problems requiring exponential time (preliminary report),” Proc. Fifth Annual ACM Symposium on Theory of Computing, STOC ’73, pp.1–9, 1973.

[19] N. Suzuki, Y. Fukushima, and K. Ikeda, “Satisfiability of simple XPath fragments under duplicate-free DTDs,” IEICE Trans. Inf. & Syst., vol.E96-D, no.5, pp.1029–1042, 2013.

[20] N. Suzuki, T. Okada, and Y. Kwon, “Detecting unsatisfiable css rules in the presence of ids,” Proc. 17th ACM SIGPLAN International Symposium on Database Programming Languages, pp.18–29, 2019.

[21] N. Walsh, “DocBookCssStylesheets,” https://github.com/docbook/wiki/wiki/DocBookCssStylesheets/.

[22] X. Zhang and J. Van den Bussche, “On the satisfiability problem
for SPARQL patterns,” Journal of Artificial Intelligence Research, vol.56, pp.403–428, 2016.

Appendix: DFA Construction for Specific Selectors

Let \( r = \text{sel} \ p : v \) be a CSS rule such that \( \text{sel} \) is specific. We show that \( \text{sel} \) can be converted into a DFA equivalent to \( \text{re}(\text{sel}) \) in polynomial time. We firstly represent \( \text{sel} \) using an equivalent regular expression. As shown in (3), \( \text{sel} \) can be divided as follows:

\[
\text{sel} = \text{sel}_1 \text{sel}_2 \cdots \text{sel}_m.
\]

Here, \( \text{sel}_i \) can be denoted

\[
\text{sel}_i = c_1 s_1 c_2 s_2 \cdots c_n s_n,
\]

where

- \( c_1 = \varepsilon \) if \( i = 1 \) and \( c_1 \in \{\omega, \neg, \} \) otherwise, and \( c_j \in \{\varepsilon, +\} \)
- \( s_j \in \Sigma \) for \( 1 \leq j \leq n \), and
- \( s_j \in \Sigma \) for \( 1 \leq j \leq n \) and \( s_j \neq s_k \) whenever \( j \neq k \).

Then \( \text{re}(\text{sel}_i) \) can be denoted as follows.

\[
\text{re}(\text{sel}_i) = S_i a_1 a_2 \cdots a_n,
\]

where

\[
S_i = \{ \Sigma_\varepsilon \quad \text{if} \quad c_1 = \varepsilon, \\
\Sigma^* \quad \text{if} \quad c_1 = \omega, \neg,
\]

and

\[
a_j = \{ (s_j)_0 \quad \text{if} \quad c_j \in \{\omega, +\}, \\
s_j \quad \text{if} \quad c_j \in \{\varepsilon, +\}. \quad (1 \leq j \leq n_i)
\]

To obtain a DFA equivalent to \( \text{re}(\text{sel}) \), we construct DFAs representing \( \text{re}(\text{sel}_i) \) for each \( i \) and merge them into a single DFA. For \( \text{re}(\text{sel}_i) = S_i a_1 a_2 \cdots a_n \), if \( a_{j-1} \in \Sigma \) and \( a_j \in \Sigma_\varepsilon \), then we say that \( a_j \) is a pivot. Similarly, if \( a_{j-1} \in \Sigma_\varepsilon \) and \( a_j \in \Sigma \), then we say that \( a_j \) is a pivot. We have the following two cases:

- **The case where \( \text{re}(\text{sel}_i) \) contains no pivot:** In this case, \( \text{re}(\text{sel}_i) \) can be represented by the DFA shown in Fig. A.1 (a), where \( \Sigma' = \Sigma \) if \( S_i = \Sigma_\varepsilon \), and \( \Sigma' = \Sigma^* \) if \( S_i = \Sigma^* \). The bottom-right edge from \( q_n \) to \( q_f \) is “optional.” More precisely, this edge is required only if \( S_i \neq S_{i+1} \) or \( i = n \). It is easy to verify that the DFAs in the figure are equivalent to \( \text{re}(\text{sel}_i) \).

- **The case where \( \text{re}(\text{sel}_i) \) contains a pivot:** Let \( a_j \) be the first pivot. In this case, \( \text{re}(\text{sel}_i) \) can be represented by the DFA shown in Fig. A.1 (b). The only difference from the above case is that for any state \( q_k \) with \( k \geq j \), the transition from \( q_k \) to \( q_1 \) and the transition from \( q_k \) to \( q_f \) are dropped. The reason is as follows. The pivot \( a_j \) does not contained in \( S_i \). Therefore, from \( a_j, a_{j+1}, \ldots, a_n \) we need not to return to the Kleene closure represented by \( S_j \).

One can see that the above automaton is deterministic since \( \text{sel} \) is specific. Let \( M(\text{sel}_i) \) be the DFA obtained from \( \text{re}(\text{sel}_i) \) as shown above. The first state \( q_0 \) of \( M(\text{sel}_i) \) is called the **start state** and the last state \( q_m \) is called the **end state**. From \( M(\text{sel}_1), M(\text{sel}_2), \ldots, M(\text{sel}_m) \), DFA \( M(\text{sel}) \) is obtained as follows.

- Merge the end state of \( M(\text{sel}_{i-1}) \) and the first state of \( M(\text{sel}_i) \) into one state for \( 2 \leq i \leq m \).
- The initial state of \( M(\text{sel}) \) is the start state of \( M(\text{sel}_1) \).

Fig. A.1  DFAs \( M(\text{sel}_i) \) constructed from \( \text{re}(\text{sel}_i) \). (a) in the case where \( \text{re}(\text{sel}_i) \) contains no pivot and (b) in the case where \( \text{re}(\text{sel}_i) \) contains a pivot.
and the accepting state of $M(\text{sel})$ is the end state of $M(\text{sel}_n)$.

For example, consider $\text{sel} = a - b + c > d$. Then $\text{sel}_1 = a$ and $\text{sel}_2 = -b + c > d$, and thus $re(\text{sel}_1) = \Sigma^*a_b$ and $re(\text{sel}_2) = \Sigma^*bc$. As shown in Fig. A.2, $M(\text{sel})$ is obtained by merging $M(\text{sel}_1)$ and $M(\text{sel}_2)$. In general, it is easy to see that $M(\text{sel})$ is equivalent to $re(\text{sel})$. 

Nobutaka Suzuki received his B.E. degree in information and computer sciences from Osaka University in 1993, and his M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology in 1995 and 1998, respectively. He was with Okayama Prefectural University as a Research Associate in 1998–2004. In 2004, he joined University of Tsukuba as an Assistant Professor. Since 2020, he has been a Professor of Faculty of Library, Information and Media Science, University of Tsukuba. His current research interests include database theory and structured documents.

Yeondae Kwon received her M.S. degree in biochemistry from Pusan National University, Busan, Korea, and Ph.D. degree in information science from Nara Institute of Science and Technology, Nara, Japan, in 2000. She was with the University of Tokyo as a Project Associate Professor in 2017–2019. Since 2019, she has been a researcher of Research Center for Agricultural Information Technology, National Agriculture and Food Research Organization. Her current research interests include text mining and sentiment analysis with machine learning. She is a member of JSBi.

Takuya Okada received his bachelor’s degree in library and information science from University of Tsukuba in 2017, and his M.E. degree in information science from University of Tsukuba in 2019. Currently, he has been working in Nihon Unisys, Ltd. His research interests are CSS and Web data management.