Reachability Analysis of the HTML5 Parser Specification and its Application to Compatibility Testing

Yasuhiko Minamide and Shunsuke Mori

University of Tsukuba, Japan

Abstract. A draft standard for HTML, HTML5, includes the detailed specification of the parsing algorithm for HTML5 documents, including error handling. In this paper, we develop a reachability analyzer for the parsing specification of HTML5 and automatically generate HTML documents to test compatibilities of Web browsers. The set of HTML documents are extracted using our reachability analysis of the statements in the specification. This analysis is based on a translation of the specification to a conditional pushdown system and on a new algorithm for the reachability analysis of conditional pushdown systems.
In our preliminary experiments, we generated 353 HTML documents automatically from a subset of the specification and found several compatibility problems by supplying them to Web browsers.

1 Introduction

A draft standard for HTML, HTML5 [Con12], includes the detailed specification of the parsing algorithm for HTML5 documents, including error handling. Although it is intended that this will solve compatibility issues in HTML parsing, several current implementations of Web browsers and parsing libraries have compatibility issues caused by the complexity of the specification.

In this paper, we develop an analyzer for the specification that checks the reachability of statements in the specification. We then apply the analyzer to generate a set of HTML documents automatically, which are used to test compatibilities of Web browsers with respect to the specification. The set of generated HTML documents enables path testing. That is, the tests cover both true and false cases for all conditional statements in the specification. The reachability analysis is based on a translation of the specification to a *conditional pushdown system* [LO10] and on a new algorithm for the reachability analysis of conditional pushdown systems.

In the first step of the development, we introduce a specification language to describe the parsing algorithm of HTML5 formally. We concentrate on the stage of parsing that follows tokenization, called tree construction in the specification. The algorithm for the tree-construction stage is specified in terms of a stack machine, with the behaviour for each input token being described informally

in English. We formalize the specification by introducing an imperative programming language with commands for manipulating the stack. The distinctive feature of the specification is that the specification inspects not only the top of the stack, but also the contents of the whole stack. Furthermore, the parsing algorithm destructively modifies the stack for elements called formatting elements. However, we exclude formatting elements from our formalized specification because of difficulties with the destructive manipulation of the stack. This is the main limitation of our work.

Our reachability analysis of the specification is based on that for conditional pushdown systems. Esparza *et al.* introduced pushdown systems with checkpoints that has the ability to inspect the contents of the whole stack and showed that they can be translated to ordinary pushdown systems [EKS03]. Li and Ogawa reformulated their definition and called them conditional pushdown systems [LO10]. The translation to ordinary pushdown systems causes the size of the stack alphabet to increase exponentially, which makes direct translation infeasible for the implementation of the reachability analysis. To overcome this problem, we design a new algorithm for reachability analysis that is a direct extension of that for ordinary pushdown systems [BEM97,EHRS00]. Our algorithm is obtained by extending $\mathcal{P}$-automata that describe a set of configurations to automata with regular lookahead. Although it still has an exponential complexity, it avoids the exponential blowup caused by the translation before applying the reachability analysis.

We have developed a reachability analyzer for the HTML5 parser specification based on the translation to a conditional pushdown system and on the reachability analysis on it. A nontrivial subset of the tree-construction stage consisting of 24 elements and 9 modes is formalized in our specification language. In our preliminary experiments, we have generated 353 HTML documents automatically from the subset of the specification and found several compatibility problems by supplying them to Web browsers.

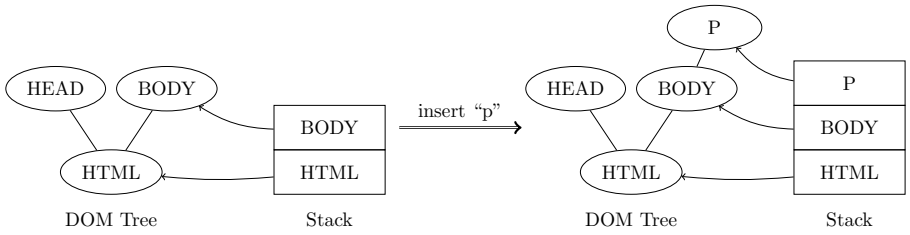
This paper is organized as follows. Section 2 reviews the HTML5 parser specification and introduces our language for formalizing the specification. The reachability analysis of the specification and its application to test-case generation are also discussed in this section. We introduce conditional pushdown systems and present a new algorithm for their reachability analysis in Section 3. The translation from the specification language to conditional pushdown systems is described in Section 4. In Section 5, we describe our implementation and present our experimental results. Finally, we discuss related work and conclude.

2 HTML5 Parser Specification and Reachability Analysis

2.1 HTML5 Parser Specification

The algorithm for the HTML5 parsing is specified as a stack machine whose behaviour depends on a variable called the *insertion mode*. The insertion mode keeps track of the part of an HTML document that the parser is processing, such

as “initial”, “in body”, or “in table”. The *stack of open elements* stores elements that have not yet been closed during parsing, and is used to match corresponding end tags and to handle errors. When the parser inserts a new HTML element, it appends the new element to the top element of the stack and pushes it onto the stack as follows.



The specification is written in English and is quite complex. The following is part of the HTML5 specification for the “in body” insertion mode.

> ↪ *A start tag whose tag name is one of: ..., “p”, ...*
> *If the stack of open elements has a p element in button scope, then act as if an end tag with the tag name “p” had been seen.*
> *Insert an HTML element for the token.*

The specification is sometimes rather difficult to interpret precisely, and it is not possible to analyze the specification mechanically.

The first step in the analysis of the specification is to introduce a specification language. Figure 1 is an example of a formalized specification using our specification language. The specification comprises a set of mode definitions, with each mode definition containing specifications of the behaviour for start and end tags in the mode. The behaviour for each tag is described as an imperative program that manipulates the stack of open elements with commands including `PUSH` and `POP`. We also allow the following commands:

- `MODE[mode]` changes the insert mode to `mode`. The change of mode affects the behaviour of the `PSEUDO` command below.
- `PSEUDO[t]` is basically a procedure call and the parser acts as if a tag `t` had been seen.
- `ERROR` records that an error is encoutered during parsing and does nothing in our model.

In the specification for each tag, the variable `me` refers to the element name for the tag. The command `insertElement` is currently defined as follows.

```
sub insertElement [target] = PUSH [target]
```

This definition is used because we are currently interested only in the reachability analysis of the specification and are ignoring the construction of the DOM tree.

The most notable feature of the specification language is the inspection of the current stack content. In the example, the current contents of the stack

```
mode inbody{
  <p> : {
    if isInScope[ buttonScopeElements, {P} ] then
      PSEUDO[</p>];
    insertElement[me]
  }
  <h1>, <h2> : {
    ...
    if match[ {H1 | H2} .* ] then{
      ERROR; POP
    };
    insertElement[me]
  }
  <table> : {
    insertElement[{Table}];
    MODE["intable"]
  }
  ...
}
```

Fig. 1. Example of a formalized specification

The stack of open elements is said to have an element in a specific scope consisting of a list of element types list when the following algorithm terminates in a match state:

1. Initialize node to be the current node (the bottommost node of the stack).
2. If node is the target node, terminate in a match state.
3. Otherwise, if node is one of the element types in list, terminate in a failure state.
4. Otherwise, set node to the previous entry in the stack of open elements and return to step 2. (This will never fail, since the loop will always terminate in the previous step if the top of the stack — an html element — is reached.)

Fig. 2. The algorithm of "have a element in a specific scope"

are inspected by a regular expression `match[ {H1 | H2} .* ]`, where regular expression `{H1 | H2} .*` represents stacks whose top element is either `H1` or `H2`.

The real capability stack inspection is utilized in the definition for `<p>` as `isInScope[ buttonScopeElements, {P} ]`. It is the formalization of "have an element in a specific scope" and its specification is shown in Figure 2. Although the algorithm is rather complicated, the property can be checked by the following inspection of the stack using a regular expression:

```
fun isInScope [list,target] = match[ (element \ list)* target .* ]
```

where `element` is a variable representing the set of all elements, and therefore the set `element \ list` contains elements that exclude those in `list`.

In the formalization of the HTML5 specification, we also make explicit some of the implicit assumptions that appear in the specification. In the following

```
mode inbody {
 <select> : { PUSH[ me ]; MODE["inselect"] }
}
mode inselect {
 <option>, <optgroup> : { PUSH[ me ] }

 </optgroup> : {
    if match[ {Option} {Optgroup} .* ] then
      POP    // (A)
    else
      NOP;   // (B)
    if match[ me .* ] then
      POP    // (C)
    else
      ERROR  // (D)
   }
}
```

Fig. 3. Example for reachability analysis

example, it is assumed that, at this point, the stack of open elements will have either a "td" or "tr" element in the table scope.

> *If the stack of open elements has a td element in table scope, then act as if an end tag token with the tag name "td" had been seen.*
> *Otherwise, the stack of open elements* ***will have a th element in table scope****; act as if an end tag token with the tag name "th" had been seen.*

We formalize this specification by using the command `FATAL` and show that `FATAL` is not reachable by applying our reachability analyzer. Please note that it is normal to reach an `ERROR` command because it just records the parser encounter an ill-formed HTML document.

```
if isInTableScope [ { Td } ] then ...
else if isInTableScope [ { Th } ] then ...
else FATAL
```

2.2 Reachability Analysis and Test Generation

We analyze the reachability of specification points via translation to a conditional pushdown system. The main application is to test the compatibility of HTML5 parsing with Web browsers and parsing libraries. Our reachability analyzer generates test cases that cover both true and false cases for all conditional statements in the specification.

Let us consider the example shown in Figure 3. To cover both true and false cases for all conditional statements, our reachability analyzer must check the reachability of the points (A)–(D). By translating the specification to a conditional pushdown system and applying the reachability analysis described in Section 3.2, the analyzer finds that the point (A) is reachable from the initial state of inbody with the empty stack by the following input.

```
<select><optgroup><option></optgroup>
```

A test document is generated from this input by appending appropriate end tags as follows. By executing the interpreter of the specification language, we compute the stack after the execution for the above input. Before the execution of (A), we have stack `Option, Optgroup, Select`. The pop statements at (A) and (C) are then executed. The execution for the input then results in stack `Select`. We therefore generate the following HTML document as a test case by appending the end tag of `Select`.

```
<select><optgroup><option></optgroup></select>
```

By applying the same method, we obtain the following test cases for (B)–(D).

```
<select></optgroup></select>           // (B)
<select><optgroup></optgroup></select> // (C)
<select></optgroup></select>           // (D)
```

We can then test the compatibility of Web browsers by supplying test cases generated in this manner as HTML documents.

3 Conditional Pushdown Systems and Reachability Analysis

We translate our specification language into conditional pushdown systems of Li and Ogawa [LO10], which are a reformulation of pushdown systems with checkpoints [EKS03]. Conditional pushdown systems extend ordinary pushdown systems with the ability to check the contents of the whole stack against a regular language.

3.1 Regular Languages and Derivatives

We briefly review the theory of regular languages with a focus on the derivatives of regular languages [Brz64]. The theory of derivatives has drawn renewed attention as an implementation technique for parsing and decision procedures on regular languages [ORT09,KN11]. Let $\mathrm{Reg}(\Gamma)$ be the set of regular languages over Γ.

For $L \subseteq \Gamma^*$ and $w \in \Gamma^*$, the *derivative*[1] of L with respect to w is written as $w^{-1}L$ and defined as follows.

$$w^{-1}L = \{w' \mid ww' \in L\}$$

Brzozowski showed that there are a finite number of types of derivatives for each regular language. More precisely, the set $\{w^{-1}L \mid w \in \Gamma^*\}$ is finite for any regular language L over Γ. This fact is the key to the termination of our algorithm for the reachability analysis of conditional pushdown systems.

[1] The derivative $w^{-1}L$ is also called the left quotient in many litterateurs.

In this paper, regular languages are often described in terms of regular expressions. The syntax of regular expressions over Γ is defined as follows[2]:

$$R ::= \emptyset \mid \epsilon \mid \gamma \mid R \cdot R \mid R + R \mid R^*$$

where $\gamma \in \Gamma$. We write $L(R)$ for the language of regular expression R. We say that a regular expression R is nullable if $\epsilon \in L(R)$. We characterize nullable expressions in terms of following function $\nu(R)$.

$$\begin{array}{rr} \nu(\emptyset) = \emptyset & \nu(\epsilon) = \epsilon \\ \nu(\gamma) = \emptyset & \nu(R^*) = \epsilon \\ \nu(R_1 + R_2) = \nu(R_1) + \nu(R_2) & \nu(R_1 \cdot R_2) = \nu(R_1) \cdot \nu(R_2) \end{array}$$

Brzozowski showed that the derivative $\gamma^{-1}R$ of a regular expression can be computed symbolically using $\nu(R)$, as follows:

$$\begin{array}{rr} \gamma^{-1}\emptyset = \emptyset & \gamma^{-1}\epsilon = \emptyset \\ \gamma^{-1}\gamma = \epsilon & \gamma^{-1}\gamma' = \emptyset \\ \gamma^{-1}R^* = (\gamma^{-1}R)R^* & \gamma^{-1}(R_1 + R_2) = \gamma^{-1}R_1 + \gamma^{-1}R_2 \end{array}$$

$$\gamma^{-1}R_1 \cdot R_2 = (\gamma^{-1}R_1)R_2 + \nu(R_1)(\gamma^{-1}R_2)$$

The derivative of a regular expression can be extended for words with $\epsilon^{-1}R = R$ and $(\gamma w)^{-1}R = w^{-1}(\gamma^{-1}R)$. We then have $w^{-1}L(R) = L(w^{-1}R)$.

Our implementation utilizes regular expressions extended with intersection and complement. The derivatives of extended regular expressions are computed similarly, as described in [ORT09]. The automaton corresponding to a regular expression is constructed only when we decide the language inclusion between two regular expressions.

3.2 Conditional Pushdown Systems

We now review conditional pushdown systems that have the ability to check the current stack contents against a regular language, and then present a new algorithm for the reachability analysis.

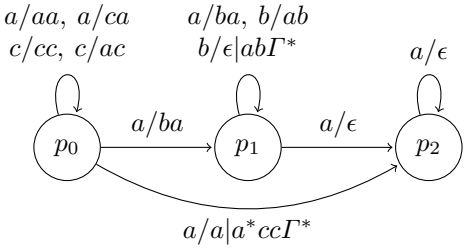
Definition 1. *A conditional pushdown system $\mathcal{P}$ is a structure $\langle P, \Gamma, \Delta \rangle$, where P is a finite set of states, Γ is a stack alphabet, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^* \times \mathrm{Reg}(\Gamma)$ is a set of transitions.*

A configuration of conditional pushdown system $\mathcal{P}$ is a pair $\langle p, w \rangle$ where $p \in P$ and $w \in \Gamma^*$. The set of all configurations is denoted by $\mathcal{C}$. We write $\langle p, \gamma \rangle \overset{R}{\hookrightarrow} \langle p', w \rangle$ if $\langle p, \gamma, p', w, R \rangle \in \Delta$. The reachability relation is defined as an extension of that for ordinary pushdown systems. A configuration $\langle p, \gamma w' \rangle$ is an immediate predecessor of $\langle p', ww' \rangle$ if $\langle p, \gamma \rangle \overset{R}{\hookrightarrow} \langle p', w \rangle$ and $w' \in R$: the

[2] For regular expressions in our specification language, we write alternation as $R_1|R_2$ instead of $R_1 + R_2$.

regular language R inspects the current stack contents excluding its top. The reachability relation $\Longrightarrow$ is the reflexive and transitive closure of the immediate successor relation. Then, the predecessor function $\mathrm{pre}^* : 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ is defined by $\mathrm{pre}^*(C) = \{c \mid \exists c' \in C.c \Longrightarrow c'\}$.

Let us consider the following conditional pushdown system $\mathcal{P}_1$ shown below. A transition labeled with $\gamma/w|R$ from p to p' denotes transition rule $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', w\rangle$, and a transition labeled with γ/w is an abbreviation of $\gamma/w|\Gamma^*$.



In the transition from p_0 to p_2, the condition $a^*cc\Gamma^*$ is used to check that two c's were pushed at p_0 consecutively. In the transition from p_1 to p_1, the condition $ab\Gamma^*$ is used to prevent the popping of the last b on the stack.

As discussed by Esparz *et al.* [EKS03], a conditional pushdown system can be translated into an ordinary pushdown system by expanding its stack alphabet. However, the translation causes the size of the stack alphabet and the transition relation to grow exponentially, and it is therefore not feasible to apply the translation for the reachability analysis directly.

3.3 New Algorithm for Reachability Analysis

We describe our new algorithm for the reachability analysis of conditional pushdown systems. The reachability analysis of ordinary pushdown systems represents a regular set of configurations with $\mathcal{P}$-automata [BEM97,EHRS00]. We directly extend the algorithm by representing a regular set of configurations with $\mathcal{P}$-automata using regular lookahead.

Given a conditional pushdown system $\mathcal{P} = \langle P, \Gamma, \Delta\rangle$, a $\mathcal{P}$-automaton uses P as a set of initial states and Γ as the input alphabet.

Definition 2. *A $\mathcal{P}$-automaton with regular lookahead is a structure $\mathcal{A} = \langle \Gamma, Q, \delta, P, F\rangle$, where Q is a finite set of states satisfying $P \subseteq Q$, $\delta \subseteq Q \times \Gamma \times Q \times \mathrm{Reg}(\Gamma)$ is a set of transition rules, and F is a set of final states.*

We introduce the transition relation of the form $q \overset{w|w'}{\longrightarrow} q$ for $\mathcal{P}$-automata with regular lookahead: it means that, at the state q, the automaton may consume w and change its state to q' if the rest of the input is w'. This is defined as follows:

- $q \overset{\epsilon|w}{\longrightarrow} q$ for any $q \in Q$ and any $w \in \Gamma^*$,
- $q \overset{\gamma|w}{\longrightarrow} q'$ if $\langle q, \gamma, q', R\rangle \in \delta$ and $w \in R$,
- $q \overset{w\gamma|w'}{\longrightarrow} q'$ if $q \overset{w|\gamma w'}{\longrightarrow} q''$ and $q'' \overset{\gamma|w'}{\longrightarrow} q'$.

Then, the set of configurations represented by $\mathcal{A}$ is defined as $\mathrm{Conf}(\mathcal{A}) = \{\langle p, w\rangle \mid p \in P \text{ and } p \xrightarrow{w|\epsilon} q \text{ for some } q \in F\}$.

To formulate our new algorithm for reachability analysis, we also extend the transition rules to those involving many steps, namely $q \xrightarrow{w|R} q'$, as follows:

- $q \xrightarrow{\epsilon|\Gamma^*} q$,
- $q \xrightarrow{\gamma|R} q'$ if $\langle q, \gamma, q', R\rangle \in \delta$,
- $q \xrightarrow{w\gamma|\gamma^{-1}R_1 \cap R_2} q'$ if $q \xrightarrow{w|R_1} q''$ and $q'' \xrightarrow{\gamma|R_2} q'$.

In the third case of the above definition, the two transition rules are combined by composing two lookahead sets via quotient and intersection: $\gamma^{-1}R_1 \cap R_2$. At the state q', $\gamma^{-1}R_1$ must be satisfied because the symbol γ is consumed by the transition from q'' to q'. The following lemma relates the extended transition rules to transitions.

Lemma 1.

- *If* $q \xrightarrow{w|R} q'$ *and* $w' \in R$, *then* $q \xrightarrow{w|w'} q'$.
- *If* $q \xrightarrow{w|w'} q'$, *then* $q \xrightarrow{w|R} q'$ *and* $w' \in R$ *for some* R.

The $\mathcal{P}$-automaton $\mathcal{A}_{\mathrm{pre}^*}$ representing $\mathrm{pre}^*(\mathrm{Conf}(\mathcal{A}))$ can be computed by extending the *saturation rule* of [BEM97]. That is, $\mathcal{A}_{\mathrm{pre}^*}$ is obtained by adding new transitions according to the following extended saturation rule:

- If $\langle p, \gamma\rangle \overset{R_1}{\hookrightarrow} \langle p', w\rangle$ and $p' \xrightarrow{w|R_2} q$ in the current automaton, add a transition rule $p \xrightarrow{\gamma|R_1 \cap R_2} q$.

Based on this saturation rule, we have also extended the efficient algorithm for the reachability analysis [EHRS00] in a straightforward manner.

The following lemma and the finiteness of derivatives of a regular language guarantee the termination of the application of the saturation rule.

Lemma 2. *Let* $\mathcal{R} = \{R \mid \langle q, \gamma, q', R\rangle \in \delta$ *for some* q, γ, $q'\}$.
If $q \xrightarrow{w|R} q''$, *then* $R = \bigcap \mathcal{R}'$ *for some* $\mathcal{R}' \subseteq \{w^{-1}R \mid R \in \mathcal{R} \wedge w \in \Gamma^*\}$.

Let us consider the previous conditional pushdown system $\mathcal{P}_1$. We apply the saturation algorithm to $\mathcal{P}_1$ to check the reachability to the set of configurations $C_1 = \{\langle p_2, w\rangle \mid w \in L(c(a+c)^*b\Gamma^*)\}$. The $\mathcal{P}_1$-automaton in Figure 4 excluding the dashed transitions represents C_1 by using lookahead. The dashed transitions are added by applying the saturation rule. The three transitions from p_0 to q_f are added from the top. This shows that the configuration C_1 is reachable from p_0 with a stack satisfying $(a+c)^+b\Gamma^*$.

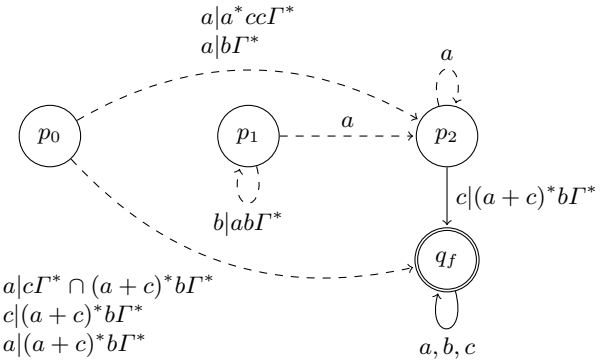


Fig. 4. $\mathcal{P}_1$-automaton obtained by the saturation algorithm

4 Translation to Conditional Pushdown Systems

In this section, we present the translation of the specification language to conditional pushdown systems.

4.1 Expanding Pseudo Statements

The first step of the translation is to expand pseudo statements `PSEUDO[t]` at non-tail positions. This is necessary because `PSEUDO[t]` is basically a procedure call and its simulation requires another stack that is not synchronized with the stack of open elements. Pseudo inputs at tail positions can be translated directly into transitions of a pushdown automaton. To avoid infinite chains of inline expansion, we do not expand `PSEUDO[t]` inside the code for the tag `t`. In the following example, `PSEUDO[</p>]` in `<p>` and `PSEUDO[<p>]` in `</p>` should be expanded because they are not at tail positions.

```
<p> : {
  if isInButtonScope[ {P} ] then
    PSEUDO[</p>];
  insertElement[me]
}
</p> : {
  if !isInButtonScope [ {P} ] then {
    PSEUDO[<p>];  PSEUDO[</p>]
  } else {
    popuntil[{P}]
  }
}
```

We obtain the following code by expanding them. Because we cannot expand `PSEUDO[<p>]` in the code for `<p>`, it is translated into `FATAL`. If the `FATAL` introduced in this translation is reachable, then the translation will not be faithful. However, this is not the case in this example because `PSEUDO[<p>]` is constrained by `isInButtonScope[ {P} ]` and `!isInButtonScope[ {P} ]`.

```
<p> : {
  if isInButtonScope[ {P} ] then {
    if !isInButtonScope [ {P} ] then {
      PSEUDO[<p>];  => FATAL
      PSEUDO[</p>]
    } else {
      popuntil[{P}]
    }
  }
  insertElement[me]
}
...
```

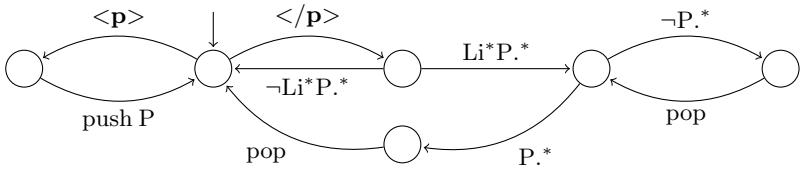
In order to check that `FATAL` statements introduced by this expansion are not reachable, we check their reachability after the translation to a conditional pushdown system. For the subset of the HTML5 specification we have formalized, our reachability analyzer showed that they are not reachable.

4.2 Translation to Conditional Pushdown Systems

After expansion of pseudo statements, a specification is translated to a conditional pushdown automaton. Then, a conditional pushdown system is obtained by forgetting the input of the pushdown automaton. Let us consider the following specification as an example.

```
</p> : {
  if match[ {Li}*{P}.* ] then {
    while !match[{P} .*] do POP; POP
  }
}
<p> : { PUSH[{P}] }
```

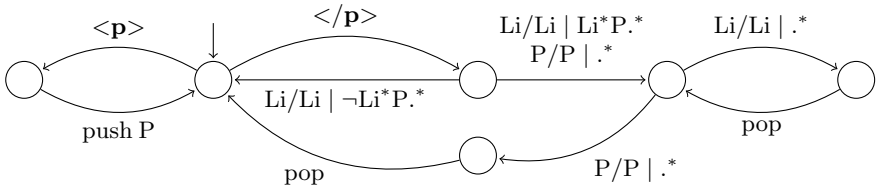
This can be converted into the following state transition diagram, where each transition is labeled with a tag indicating input, push, pop, or the condition under which the transition occurs.



The transitions labeled with a tag, push, or pop can be translated directly to those of pushdown automata. The transition label with a regular expression representing the condition under which it occurs is translated as follows. Let us consider a transition labeled with a regular expression R from q to q'.

- For each γ in the stack alphabet, a transition $\langle q, \gamma \rangle \overset{\gamma^{-1}R}{\hookrightarrow} \langle q', \gamma \rangle$ is added to the pushdown automaton.

In this example, the previous state transition diagram is translated to the following conditional pushdown automaton under the stack alphabet $\{\mathrm{P}, \mathrm{Li}\}$:



where push and pop are not translated for simplicity. The following derivatives are used in the translation.

$$\mathrm{Li}^{-1}(\mathrm{Li}^*\mathrm{P}.^*) = (\mathrm{Li}^{-1}(\mathrm{Li}^*))\mathrm{P}.^* + \nu(\mathrm{Li}^*)\mathrm{Li}^{-1}(\mathrm{P}.^*) = \mathrm{Li}^*\mathrm{P}.^*$$
$$\mathrm{P}^{-1}(\mathrm{Li}^*\mathrm{P}.^*) = (\mathrm{P}^{-1}(\mathrm{Li}^*))\mathrm{P}.^* + \nu(\mathrm{Li}^*)\mathrm{P}^{-1}(\mathrm{P}.^*) \ = .^*$$

5 Experimental Results

We have implemented the reachability analyzer of our specification language. It performs the translation from the specification language to conditional pushdown systems and reachability analyses for these systems. It is implemented in OCaml and based on the library for automata used in the PHP string analyzer [Min05].

The main application of the analyzer is to generate automatically a set of HTML test documents from a specification. It is also used to check the consistency of the specification and the translation. The current implementation checks for consistency in the following two respects.

- The execution of the specification cannot cause stack underflow.
- `FATAL` statements in the specification or introduced by the translation are unreachable.

The analyzer showed that these properties hold for the subset of the HTML5 parser specification described below.

We have formalized a nontrivial subset of the tree-construction stage of the HTML5 specification. It is 438 lines in length, excluding comments and empty lines, and contains the specification of 24 elements and 9 modes. This specification can be obtained from `http://www.score.cs.tsukuba.ac.jp/~minamide/html5spec/model.html5`. As we mentioned in the Introduction, this subset excludes the specification of formatting elements, which is one of the main limitations of our work to date.

We applied our reachability analyzer to the specification using a Linux PC with an Intel Xeon processor (3.0 GHz) and 16 GB memory. The specification is translated to a conditional pushdown automaton with 487 states, and there are 1186 specification points[3] whose reachability had to be checked. For these points, our reachability analyzer showed that 828 points were reachable from the

[3] In the implementation, a specification point is represented by a pair comprising a state and a regular expression. We may therefore have more specification points than states.

initial state and generated 353 HTML documents excluding duplicates. In the following table, the first row shows the length of an input sequence of tags and the second row shows the number of points. For example, there are 380 points for which the analyzer found an input sequence of length 3.

Length	1	2	3	4	5	6
# Points	46	167	380	198	35	2

The reachability of the specification points was checked by applying the algorithm described in Section 3.2 by adding the final states corresponding to them in the $\mathcal{P}$-automaton. It took 82 minutes to check the reachability of all the points and required more than 3 Gbyte of memory during the computation.

We conducted compatibility tests on the following Web browsers and HTML5 parser libraries: html5lib [htm] is implemented in Python and closely follows the specification, and htmlparser, the Validator.nu HTML parser [Val], is implemented in Java and has been used for HTML5 in the W3C markup validation service. The experiment was conducted on Mac OS X, version 10.7.3. The following table shows the number of incompatibilities found when using the generated set of 353 HTML documents. The numbers in parentheses are obtained after merging similar incompatibilities.[4]

	Safari	Firefox	Opera	IE [5]	html5lib	htmlparser
Version	5.1.3	10.0.1	11.61	-	0.95	1.3.1
# Incompatibilities	1 (1)	6 (2)	0 -	0 -	3 (1)	6 (2)

The three main incompatibilities found in this experiment are listed below. The lines labeled 'Test' and 'Spec' are the HTML documents generated by our analyzer and the serialized representation of the results of parsing with the HTML5 specification, respectively. The incompatible results are shown following the Spec lines.

```
Test : <body><dd><optgroup><dd></dd></body>
Spec : <body><dd><optgroup></optgroup></dd><dd></dd></body>
       Safari, Opera, html5lib, IE
     : <body><dd><optgroup><dd></dd></optgroup></dd></body>
       Firefox, htmlparser
Test : <body><ruby><button><rp></rp></button></ruby></body>
Spec : <body><ruby><button><rp></rp></button></ruby></body>
       Opera, html5lib, IE
     : <body><ruby><button></button><rp></rp></ruby></body>
       Safari, Firefox, htmlparser
Test : <body><table><li><li></li></table></body>
Spec : <body><li></li><li></li><table></table></body>
       Safari, Firefox, Opera, htmlparser, IE
       <body><li></li><table><li></li></table></body>
       html5lib
```

[4] Some of the incompatibilities are caused by differences between versions of the HTML5 specification, which is discussed below.

[5] Consumer Preview, version 10.0.8250.0.

We have investigated the second case for Firefox. The specification for the start `rp` tag can be written as follows.

```
if isInScope[{Ruby}] then {
  generateImpliedEndTag[];
  if !match[ {Ruby} .* ] then ERROR
};
insertElement[me]
```

The code of Firefox does not correspond to this, but to the specification below. We found that this is compatible with the latest *published version* of the specification, W3C Working Draft 25 May 2011, although we were working with the Editor's Draft 22 February 2012.

```
if isInScope[{Ruby}] then {
  generateImpliedEndTag[];
  if !match[ {Ruby} .* ] then ERROR
  while !match[ {Ruby} .*] do POP;    <== Extra code in Firefox
};
insertElement[me]
```

6 Related Work

The reachability analysis of pushdown systems with checkpoints was studied by Esparza *et al.* as an application of LTL model checking of pushdown systems with regular valuations [EKS03]. They presented a translation to ordinary pushdown systems. Although reachability can be decided via the translation, it is not practical to apply the translation because of exponential blowup of the size of pushdown systems. They also showed that the reachability problem of pushdown systems with checkpoints is EXPTIME-complete.

Reachability can also be decided by translation to extensions of pushdown systems such as alternating pushdown systems and stack automata [GGH67]. An analysis for alternating pushdown systems is given in [BEM97] and that for stack automata is given in [HO08] as reachability analysis for higher-order pushdown systems. Although the translations to those systems do not incur exponential blowup, their algorithms are more complicated than our reachability analysis for conditional pushdown systems. An efficient algorithm for alternating pushdown systems was developed in [SSE06,Suw09]. However, only an algorithm for a restricted class with polynomial time complexity was implemented.

7 Conclusions

We have developed a reachability analyzer for the HTML5 parser specification based on the analysis of conditional pushdown systems. The analysis is applied to the automated generation of HTML documents for path testing of the specification. Several compatibility issues in Web browsers and HTML5 parsing libraries are found by supplying the documents to them.

One of the limitations of our work is that we cannot handle the specification for formatting elements. This is because their specification requires destructive manipulation of the stack. We are planning to address this limitation by checking the reachability to the first point where a destructive operation on the stack is required.

References

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: application to model-checking. In *CONCUR '97*, pages 135–150, 1997. LNCS 1243.

[Brz64] Janusz Brzozowski. Derivatives of regular expressions. *J. ACM*, 11:481–494, 1964.

[Con12] World Wide Web Consortium. HTML5: Editor's draft 22 February 2012, 2012. http://dev.w3.org/html5/spec/Overview.html.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV 2000*, pages 232–247, 2000. LNCS 1855.

[EKS03] J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2):355–376, 2003.

[GGH67] S. Ginsburg, S. A. Greibach, and M. A. Harrison. Stack automata and compiling. *J. ACM*, 14(1):172–201, 1967.

[HO08] M. Hague and C.-H. L. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. *Logical Methods in Computer Science*, 4:1–45, 2008.

[htm] html5lib. html5lib. http://code.google.com/p/html5lib/.

[KN11] Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, published online March 2011.

[LO10] Xin Li and Mizuhito Ogawa. Conditional weighted pushdown systems and applications. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 141–150, 2010.

[Min05] Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441. ACM Press, 2005.

[ORT09] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. of Functional Programming*, 19:173–190, 2009.

[SSE06] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In *ATVA 2006*, pages 141–153, 2006. LNCS 4218.

[Suw09] Dejvuth Suwimonteerabuth. *Reachability in Pushdown Systems: Algorithms and Applications*. PhD thesis, Technischen Universität München, 2009.

[Val] Validator.nu. The validator.nu html parser. http://about.validator.nu/htmlparser/.