Validation and Boolean operations for Attribute-Element Constraints

Haruo Hosoya
Kyoto University
hahosoya@kurims.kyoto-u.ac.jp

Makoto Murata
IBM Tokyo Research Laboratory
mmurata@trl.ibm.co.jp

October 1, 2002

Abstract

Algorithms for validation and boolean operations play a crucial role in developing XML processing systems involving schemas. Although much effort has previously been made for treating *elements*, very few studies have paid attention to *attributes*. This paper presents a validation and boolean algorithms for Clark's attribute-element constraints. Although his mechanism has a prominent expressiveness and generality among other proposals, treating this is algorithmically challenging since naive approaches easily blow up even for typical inputs. To overcome this difficulty, we have developed (1) a two-phase validation algorithm that uses what we call *attribute-element automata* and (2) intersection and difference algorithms that proceed by a "divide-and-conquer" strategy.

1 Introduction

XML [3] and its predecessor, SGML [19], provides two major mechanisms for representing information: *attributes* and *elements*. There has been much debate (e.g., [25]) about attributes versus elements for the past ten years. However, this debate has reached no conclusions, and we continue to have both elements and attributes. In fact, many XML-based languages (e.g., XHTML and SVG) use both of them quite heavily.

Schema languages for XML have evolved by increasing expressiveness, allowing finer- and finer-grained controls to the structure of documents. At first, the designers of schema languages (DTD [3], W3C XML Schema [10], DSD [20], and RELAX [23]) mainly paid attention to elements. On the other hand, the treatments of attributes have been rather simplistic in early schema languages.

Recently, James Clark, in his schema language TREX [7], proposed a description mechanism for attributes that has a comparable power to existing ones for elements.

The kernel of Clark's proposal is a uniform and symmetric mechanism for representing constraints on elements and those on attributes. We call this mechanism *attribute-element constraints*. The expressive power yielded by attribute-element constraints is quite substantial. For example, by using this mechanism, we can specify a constraint that allows different subelements, depending on attribute values. Such a situation may arise when we want to specify different permissible subelements for `<div class="chapter">` and `<div class="section">` where the former case allows `section` in the subelements while the latter case does not. As another example, we can specify that either an attribute or element is used to represent some piece of information. This ability would be needed when we want to allow an attribute `name` to represent a simple string name *or* an element `name` to represent a composite value of `first` and `last` names.

Although the expressiveness of attribute-element constraints has already been established [6], algorithmic studies needed for the practical uses are yet to be done. In this paper, we present our algorithms for (1) validation and (2) boolean operations that treat Clark's attribute constraints. The importance of validation has been widely known for a long time, while that of boolean operations have just been recognized in the recent series of papers on typechecking for XML processing languages [16, 11, 26, 21, 22]. For example, intersection is used in almost all of these languages as the core of type inference mechanisms for their pattern matching facilities; difference is used in XDuce type inference for precisely treating data flow through prioritized pattern clauses [17]; and, perhaps most importantly, the inclusion test, which is difference operation followed by the emptiness test, is used for "subtyping" in many of the above-mentioned languages.

Exactly because of the ability to mix constraints for elements and those for attributes, developing above-mentioned algorithms become challenging. Naively, one may attempt to use traditional automata techniques, as usually done in the case of element-only data. This is possible since elements are ordered sequences. However, as XML defines, attributes are unordered sets; therefore the logic behind attribute constraints is quite different from traditional automata. Another attempt might be to transform attribute-element constraints so that attribute constraints are isolated from element ones. However, this turns out to blow up in common examples, as will be discussed in Section 3.2.

We have developed a validation and boolean algorithms that overcome this difficulty. Our validation algorithm uses a "two-phase" strategy. We use a variant of traditional automata that mix both attribute and element constraints. In the first phase, we validate the attributes, rewriting the automaton into one that represents only the element constraints. In the second phase, we validate the elements against the rewritten automaton. By this technique, we can achieve quite a simple validation algorithm that does not incur a blow-up. For the boolean algorithms, we adopt a "divide-and-conquer" strategy. It exploits that it is often the case that we can partition given constraint formulas into orthogonal subparts. In that case, we proceed the computation with the subparts separately and then combine the results. Although this technique cannot avoid an exponential explosion in the worst case, it appears to work well for practical inputs in our experiences.

The rest of this paper is organized as follows. The next section gives a more motivation for attribute-element constraints and basic definitions including the data model and the syntax and semantics of constraints. Section 3 and Section 4 each present the validation algorithm and the boolean algorithms. Section 5 discusses the relationship with other work and Section 6 concludes the paper. Formalizations and correctness proofs can be found in Appendix A.

2 Attribute-element constraints

In our framework, data are XML documents with the restriction that only elements and attributes can appear. That is, we consider trees where each node is given a name and, in addition, associated with a set of name-string pairs. In XML jargon, a node is called element and a name-string pair is called attribute. For example, the following is an element with name `article` that has two attributes `key` and `year` and contains four child elements—two with `authors`, one with `title`, and one with `publisher`.

```
<article key="HosoyaMurata2002" year="2002">
  <author> ... </author>
  <author> ... </author>
  <title>  ... </title>
  <publisher> ... </publisher>
</article>
```

The ordering among sibling elements is significant, whereas that among attributes is not. The same name of elements can occur multiple times in the same sequence, whereas this is disallowed for attributes.

Attribute-element constraints describe a pair of an element sequence and an attribute set. Let us illustrate the constraint mechanism. Element expressions describe constraints on elements and are regular expressions on names. For example, we can write the following expression

$$\texttt{author}^{+}\,\texttt{title}\ \texttt{publisher}?$$

to represent that a permitted sequence of child elements are one or more `author` elements followed by a mandatory `title` element and an optional `publisher` element. (Note that the explanation here is informal: for brevity, we show constraints only on names of attributes and elements. The actual constraint mechanism formalized later can also describe contents of attributes and elements.) Attribute expressions are constraints on attributes and have a notation similar to regular expressions. For example, the following expression

$$@\texttt{key}\ @\texttt{year}?$$

requires a `key` attribute and optionally allows a `year` attribute. (We prepend an at-sign to each attribute name in order to distinguish attribute names from element names.) A more complex example would be:

$$@\texttt{key}\ ((@\texttt{year}\ @\texttt{month}?)\ |\ @\texttt{date})$$

That is, we may optionally append a `month` to a `year`; or we can replace these two attributes with a `date` attribute.

Attribute expressions are different from usual regular expressions in three ways. First, attribute expressions describe (unordered) sets and therefore concatenation is commutative. Second, since names cannot be duplicated in the same set, we require expressions to permit only data that conform to this restriction (e.g., $(@\texttt{a}\,|\,@\texttt{b})\ @\texttt{a}?$ is forbidden). Third, for the same reason, repetition ($^+$) is disallowed in attribute expressions. We provide, however, "wild-card" expressions that allow an arbitrary number of arbitrary attributes from a given set of names (discussed later).

Attribute-element expressions or compound expressions allow one expression to mix both attribute expressions and element expressions. For example, we can write the following.

$$@\texttt{key}\ @\texttt{year}?\ \texttt{author}^+\ \texttt{title}\ \texttt{publisher}?$$

This expression requires both that the attributes satisfy $@\texttt{key}\ @\texttt{year}?$ and that the elements satisfy $\texttt{author}^+\ \texttt{title}\,\texttt{publisher}?$. The next example is a compound expression allowing either a `key` attribute or a `key` element, not both.

$$(@\texttt{key}\,|\,\texttt{key})\ @\texttt{year}?\ \texttt{author}^+\ \texttt{title}\,\texttt{publisher}?$$

In this way, we can express constraints where some attributes are interdependent with some elements. (Note that we can place attribute expressions anywhere—even after element expressions.) In the extreme, the last example could be made more flexible as follows

$$\begin{array}{l}(@\texttt{key}\,|\,\texttt{key})\\ (@\texttt{year}\,|\,\texttt{year})?\\ (@\texttt{author}\,|\,\texttt{author}^+)\\ (@\texttt{title}\,|\,\texttt{title})\\ (@\texttt{publisher}\,|\,\texttt{publisher})?\end{array}$$

where every piece of information can be put in an attribute or an element.

In addition to the above, we provide attribute repetition expressions, which allow zero or more attributes with arbitrary names chosen from a given set of names. In general, attribute repetitions are useful in making a schema "open" so that users can put their own pieces of information in unused attributes. For example, when we want to require `key` and `year` attributes but optionally permit any other attributes, we can write the following expression (where `(*\key\year)` represents the set of all names except `key` and `year`).

$$@\texttt{key}\ @\texttt{year}\ @\texttt{(*\textbackslash key\textbackslash year)}^*$$

Although our formulation will not include a direct treatment, attribute repetitions can be even more useful if combined with name spaces. (Name spaces are a prefixing mechanism for names in XML documents; see [2] for the details.) For example, when we are designing a schema in the name space `myns`, we can write the following to permit any attributes in difference name spaces (where `myns:*` means "any names in name space `myns`").

$$@\texttt{myns:key}\ @\texttt{myns:year}\ @\texttt{(*\textbackslash(myns:*))}^*$$

Apart from the kinds of name sets described above, the following can be useful: (1) the set of all names, (2) the set of all names in a specific name space, and (3) the set of all names except those from some specific name spaces. (In fact, these are exactly the ones supported by RELAX NG [9].)

2.1 Data model

We assume a countably infinite set $\mathcal{N}$ of *names*, ranged over by $a, b, \ldots$. We define *values* inductively as follows: a *value* v is a pair $\langle\alpha, \beta\rangle$ where

- α is a set of pairs of a name and a value, and
- β is a sequence of pairs of a name and a value.

A pair in α and a pair in β are called *attribute* and *element*, respectively. In the formalization, attributes associate names with values and therefore may contain elements. This may appear odd since XML allows only strings to be contained in attributes (and the above examples followed this). Our treatment is just for avoiding the need to introduce another syntactic category for attribute contents and thereby simplifying the formalism. We write ϵ for an empty sequence and $\beta_1\beta_2$ for the concatenation of sequences β_1 and β_2. For convenience, we define several notations for values.

$$\begin{array}{lcl}
a[v] & \equiv & \langle\emptyset, \langle a, v\rangle\rangle \\
@a[v] & \equiv & \langle\{\langle a, v\rangle\}, \epsilon\rangle \\
\langle\alpha_1, \beta_1\rangle\,\langle\alpha_2, \beta_2\rangle & \equiv & \langle\alpha_1 \cup \alpha_2, \beta_1\beta_2\rangle \\
\epsilon & \equiv & \langle\emptyset, \epsilon\rangle
\end{array}$$

For example, $@a[v]\,@b[w]\,c[u]$ means $\langle\{\langle a, v\rangle, \langle b, w\rangle\}, \langle c, u\rangle\rangle$. We write $\mathcal{V}$ for the set of all values.

2.2 Expressions

Let $\mathcal{S}$ be a set of sets of names where $\mathcal{S}$ is closed under boolean operations. In addition, we assume that $\mathcal{S}$ contains at least the set $\mathcal{N}$ of all names and the empty set $\emptyset$. Each member N of $\mathcal{S}$ is called *name set*.

We next define the syntax of expressions for attribute-element constraints. As already mentioned, our expressions describe not only top-level names of elements and attributes but also their contents (which are omitted in the examples in the beginning of this section). Since, moreover, we want expressions to describe arbitrary depths of trees, we introduce recursive definitions of expressions, that is, grammars.

We assume a countably infinite set of *variables*, ranged over by x, y, z. We use X, Y, Z for sets of variables. A *grammar* G on X is a finite mapping from X to compound expressions. Compound expressions c are defined by the following syntax in conjunction with element expressions e.

$$\begin{array}{rcl} c & ::= & @N[x]^+ \\ & & c \,|\, c \\ & & c\, c \\ & & e \\ \\ e & ::= & N[x] \\ & & \epsilon \\ & & e \,|\, e \\ & & e\, e \\ & & e^+ \end{array}$$

Note that we stratify compound expressions c and element expressions e so that expressions for attributes never appear under the repetition operator. (We treat attribute repetitions as atomic since attribute names cannot be used multiple times and therefore attribute repetitions have quite different properties from the Kleene star.) For convenience, we abbreviate $@\{a\}[x]^+$ by $@a[x]^+$ and $\{a\}[x]^+$ by $a[x]^+$. We write $\mathrm{dom}(G)$ for the domain of a grammar G. We define $\mathrm{FV}(c)$ as the set of variables appearing in c and $\mathrm{FV}(G)$ as $\bigcup_{x \in \mathrm{dom}(G)} F(x)$. We require any grammar to be "self-contained," i.e., $\mathrm{FV}(G) \subseteq \mathrm{dom}(G)$.

We define $\mathrm{elm}(c)$ as the union of all the element name sets (the N in the form $N[x]$) appearing in the expression c. Similarly, $\mathrm{att}(c)$ is the union of all the attribute name sets (the N in the form $@N[x]^+$) appearing in the expression c. We forbid expressions with overlapping attribute name sets to be concatenated. That is, any expression must not contain an expression $c_1\, c_2$ with $\mathrm{att}(c_1) \cap \mathrm{att}(c_2) \neq \emptyset$.[1]

In addition, we impose a restriction on the form $@N[x]^+$: it must be that either the name set N is singleton or the content x is a distinguished variable **any** accepting any values. We assume that any given grammar G has the following mapping.

$$G(\mathbf{any}) = \mathcal{N}[\mathbf{any}]^* \, @\mathcal{N}[\mathbf{any}]^*$$

This restriction is for ensuring closure under complementation. We discuss this in detail in Section 4.4.

By using the above syntax, we can derive the following useful forms.

$$\begin{array}{lcl} c^* & \equiv & c^+ \,|\, \epsilon \\ c? & \equiv & c \,|\, \epsilon \\ @N[x]^* & \equiv & @N[x]^+ \,|\, \epsilon \end{array}$$

[1]The stratification and the disjointness restriction on attribute names are also adopted in RELAX NG.

Our expressions do not include a "single-attribute expression" $@N[x]$ for describing a single attribute with a name from N. One may wonder if we can encode it somehow by other supported expressions. The answer is yes in the case that N is finite.[2] Indeed, if N is finite, then we can rewrite $@N[x]$ as follows, where $N = \{a_1, \ldots, a_k\}$:

$$@N[x] \quad \equiv \quad @\{a_1\}[x]^+ \mid \ldots \mid \{@a_k\}[x]^+$$

In the case that N is infinite, our framework has no way to represent single-attributes expressions. We cannot simply add single-attribute expressions with infinite name sets since it breaks closure under complementation. For example, the complementation of $@\mathcal{N}[\mathbf{any}]$ would be zero, two, or more attributes with any name and any content. However, there is no way to express "two or more" in our framework. It may appear disappointing, but such a constraint as "only one attribute is present from a given infinite set" seems hardly useful.

2.3 Semantics

The semantics of expressions with respect to a grammar is described by the relation of the form $G \vdash v \in c$, which is read "value v conforms to expression c under G." This relation is inductively defined by the following rules.

$$\frac{\forall i.\ (a_i \in N \quad G \vdash v_i \in G(x)) \qquad k \geq 1 \qquad a_i \neq a_j \text{ for } i \neq j}{G \vdash @a_1[v_1] \ldots @a_k[v_k] \in @N[x]^+} \quad (\text{T-AttRep})$$

$$\frac{a \in N \qquad G \vdash v \in G(x)}{G \vdash a[v] \in N[x]} \quad (\text{T-Elm})$$

$$\frac{G \vdash v \in c_1 \qquad \text{or} \qquad G \vdash v \in c_2}{G \vdash v \in c_1 \mid c_2} \quad (\text{T-Alt})$$

$$\frac{G \vdash v_1 \in c_1 \qquad G \vdash v_2 \in c_2}{G \vdash v_1\, v_2 \in c_1\, c_2} \quad (\text{T-Cat})$$

$$G \vdash \epsilon \in \epsilon \quad (\text{T-Eps})$$

$$\frac{\forall i.\, G \vdash v_i \in c \qquad k \geq 1}{G \vdash v_1 \ldots v_k \in c^+} \quad (\text{T-Plu})$$

Note that rules T-Alt and T-Cat treat alternation and concatenation both in compound expressions and element expressions.

[2] RELAX NG has single-attribute expressions and adopts this restriction of finiteness.

3 Validation

In this section, we present a validation technique for values containing both attributes and elements.

3.1 Attribute-free Validation

We first consider validation for attribute-free grammars. To validate values against element expressions, we introduce element automata, which are variations of usual string automata. Element automata play critical roles in most validation algorithms [24]. Later in this section, we introduce attribute-element automata by extending element automata.

An attribute-free grammar is a finite mapping from variables to element expressions. Recall that an element expression is a regular expression whose atoms are of the form $N[x]$. From each element expression, we can construct an automaton whose transition labels are also of the form $N[x]$. Formally, we first define automata in the usual way: an *automaton* M on an alphabet Σ is a tuple $(Q, q^{\text{init}}, Q^{\text{fin}}, \delta)$ where Q is a finite set of *states*, $q^{\text{init}} \in Q$ is an *initial state*, $Q^{\text{fin}} \subseteq Q$ is a set of *final states*, and $\delta \subseteq (Q \times \Sigma \times Q) \cup (Q \times Q)$ is a *transition relation* [14]. Note that we have allowed null transitions by incorporating $Q \times Q$. Then, an *element automaton* is an automaton over $\{N[x] \mid N \in S, x \in X\}$, where S is a set of name sets and X is a set of variables. Well-known algorithms for creating automata from regular expressions can be used for creating element automata from element expressions by assuming $N[x]$ as symbols.

We give a brief sketch of a validation algorithm for attribute-free grammars, although understanding of this sketch is not required to read the rest of this paper. We validate a value by assigning variables to every element (including every descendant element) in this value. The variable assignment must "conform" to the grammar. That is, for each element, let x be the variable assigned to this element and M be the element automaton constructed from the element expression defined as x in the grammar. We execute the automaton for the children of the element in such a way that each child element $a[v]$ matches a transition labeled $N[x]$ when N contains a and x is assigned to this element $a[v]$. If we can successfully assign variables to all elements, we report that the value is valid. Although this algorithm may be too naive in that it tests all possible assignments, there is a known algorithm that takes only linear time in the size of the input value. Interested readers are referred to a note [24].

3.2 Validation by separating attributes and elements

Attribute-element constraints pose a significant challenge. We cannot directly use element automata described above since our values are set-sequence pairs whereas element automata accept just sequences.

A naive solution is to separate constraints on attribute sets and those on element sequences. For example, we can handle the above compound expression by first converting it to the following expression.

$$
\begin{array}{ll}
(((@a[\epsilon]^{+}\ @b[\epsilon]^{+}) & \epsilon) \\
\mid (@a[\epsilon]^{+} & b[\epsilon]) \\
\mid (@b[\epsilon]^{+} & a[\epsilon]) \\
\mid (\epsilon & a[\epsilon]\ b[\epsilon]))
\end{array}
$$

Observe that this compound expression is a choice of c-e pairs, where c is an element-free compound expression and e is an element expression. Here c describes *sets* of attributes, while e describes *sequences* of elements. Any compound expression can be normalized to this normal form, although we do not further describe any specific normalization procedure in this paper.

After normalization, we can easily compare values against normalized compound expressions. Given a value $\langle \alpha, \beta \rangle$, we examine if, for some c-e pair, α is valid against c and β is accepted by an element automaton constructed from e. The rest of validation is the same as in attribute-free grammars.

However, this approach causes combinatory explosion easily. For example, the normalization shown above created 4 $(= 2^2)$ pairs, since one is constructed for each subset of $\{@a[\epsilon]^{+}, @b[\epsilon]^{+}\}$. If we normalize

$$((@a[\epsilon]^{+}|a[\epsilon])\,(@b[\epsilon]^{+}|b[\epsilon])\ \ldots (@y[\epsilon]^{+}|y[\epsilon])\,(@z[\epsilon]^{+}|z[\epsilon])),$$

we will have 2^{26} pairs.

3.3 Attribute-element automata

We extend our attribute-free validation technique for handling attributes as well as elements. This extension introduces three steps: (1) given a compound expression, we insert special atoms (called "non-existent attribute declaration") to it; (2) we create an automaton (called "attribute-element automaton") from the compound expression, (3) given an attribute set, we create an element automaton from this attribute-element automaton by rewriting some of its transitions. In an actual implementation, we can perform the first and second steps statically, i.e., before receiving values. On the other hand, the third step can only be done dynamically. The rest of validation is the same as in attribute-free validation.

Below, for the ease of explanation, we first present the second and third steps. If we perform these two steps only, we sometimes report invalid values valid. We then present the first step as a remedy to this problem.

The second step creates an attribute-element automaton. Recall that atoms of compound expressions are of the form $@N[x]^{+}$ or $N[x]$. We can assume that a compound expression is a regular expression whose atoms are $@N[x]^{+}$ or $N[x]$, and construct an automaton from this compound expression. Thus, some transitions of this automaton have $@N[x]^{+}$ as labels and others have $N[x]$ as labels.

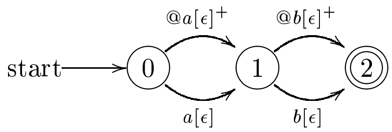
The third step creates an element automaton by rewriting some transitions of the constructed automaton. Given an attribute set α, we replace each $@N[x]$-transition by a null transition if (1) more than one attribute in α is valid against

$@N[x]^+$ and (2) no other attributes in α have names in N. Otherwise, we remove this transition. We preserve those transitions having $N[x]$ as labels.

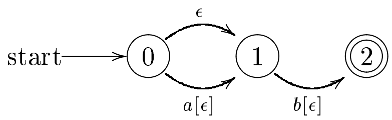
To illustrate, we use the aforementioned expression

$$(@a[\epsilon]^+ \mid a[\epsilon])\,(@b[\epsilon]^+ \mid b[\epsilon]).$$

An automaton constructed from this compound expression is shown below.



Suppose that we would like to validate $@a[\epsilon]b[\epsilon]$. Since $@a[\epsilon]$ is the only attribute in this value, we replace the transition labeled $@a[\epsilon]^+$ by a null transition, and remove the transition labeled $@b[\epsilon]^+$. The element automaton obtained by this rewriting is shown below. The only element in $@a[\epsilon]b[\epsilon]$ is $b[\epsilon]$, which is accepted by this element automaton. We can thus correctly report that $@a[\epsilon]b[\epsilon]$ is valid.



However, the element automaton also accepts $a[\epsilon]b[\epsilon]$ and we thus mistakenly report that $@a[\epsilon]a[\epsilon]b[\epsilon]$ is valid. Furthermore, $@f[\epsilon]a[\epsilon]b[\epsilon]$ (where $@f[\epsilon]$ is undeclared) is also mistakenly reported valid: although we remove the transitions for $@a[\epsilon]$ and $@b[\epsilon]$, the element automaton still accepts $a[\epsilon]b[\epsilon]$. What we missed is that, although we ensured that attributes *required* by compound expressions are *present* in values, we did not ensure that attributes *not required* by compound expressions are *absent* in values.

To overcome this problem, we introduce the first step. This step saturates a compound expression by introducing *non-existent attribute declarations*. A non-existent attribute declaration is $!@N$, where N is a name set. Informally, $!@N$ means that no attributes have names in N. When a grammar maps a variable x to some compound expression c, we prepend a non-existent attribute declaration to c, which ensures that attributes not required by c are absent. When an alternative $c_1 \mid c_2$ of two compound expressions occurs in c, we prepend a non-exitent attribute declaration to c_1, which ensures that attributes required by c_2 only are absent. Likewise, we prepend another non-existent attribute declaration to c_2.

We accordingly modify the second and third steps. An element-attribute automaton constructed by the second step has transitions having $!@N$ as labels. The third step has to rewrite these transitions. Given an attribute set α, we replace each $!@N$-transition by a null transition if no attributes in α have names in N. Otherwise, we remove this transition.

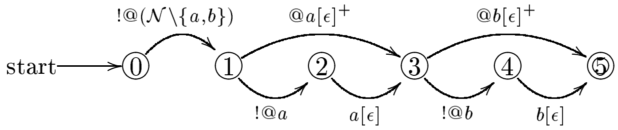
As an example, we again consider

$$(@a[\epsilon]^+ \mid a[\epsilon])\,(@b[\epsilon]^+ \mid b[\epsilon]).$$

By introducing non-existent attribute declarations, we saturate it as follows:

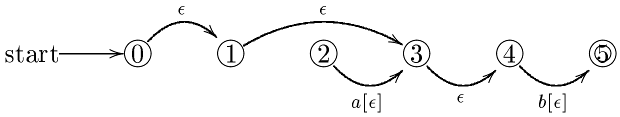
$$!@(\mathcal{N} \setminus \{a, b\})\, (@a[\epsilon]^+ \mid (!@a\; a[\epsilon]))\, (@b[\epsilon]^+ \mid (!@b\; b[\epsilon])),$$

where $\mathcal{N} \setminus \{a, b\}$ denotes the set of names except a and b. From the saturated expression, we create an automaton shown below.



Now, this automaton has transitions labeled $@a[\epsilon]^+$ and $@b[\epsilon]^+$ as well as transitions labeled $!@a$, $!@b$, and $!@(\mathcal{N} \setminus \{a, b\})$. We call such automata *attribute-element automata*.

Suppose that we want to validate $@a[\epsilon]a[\epsilon]b[\epsilon]$. As previously, we replace the transition labeled $@a[\epsilon]^+$ by a null transition and remove the transition labeled $@b[\epsilon]^+$. We further replace the transition labeled $!@b$ by a null transition, since there are no attributes named b. We also replace the transition labeled $!@(\mathcal{N} \setminus \{a, b\})$ by a null transition, since no attributes have names other than a or b. But we remove the transition labeled $!@a$, since there is an attribute named a. The element automaton obtained by this rewriting (shown below) accepts $b[\epsilon]$ only. Thus, we do not mistakenly report that $@a[\epsilon]a[\epsilon]b[\epsilon]$ is valid.



Let us make sure that undeclared attributes lead to invalidity. Suppose that a given value has an attribute $@f[\epsilon]$. Then, the transition labeled $!@(\mathcal{N} \setminus \{a, b\})$ is removed. Since this transition is the only transition from the start state, the created element automaton accepts no element sequences.

We now formally describe the three steps.

The first step is effected by a saturating operator. This operator introduces non-existent attribute declarations for each alternative of compound expressions.

$$\begin{array}{lcl}
\mathrm{sat}(@N[x]^+) & = & @N[x]^+ \\
\mathrm{sat}(c_1 \mid c_2) & = & (!@(\mathrm{att}(c_2) \setminus \mathrm{att}(c_1))\;\; \mathrm{sat}(c_1)) \\
 & & \quad \mid \;\; (!@(\mathrm{att}(c_1) \setminus \mathrm{att}(c_2))\;\; \mathrm{sat}(c_2)) \\
\mathrm{sat}(c_1\, c_2) & = & \mathrm{sat}(c_1)\, \mathrm{sat}(c_2) \\
\mathrm{sat}(N[x]) & = & N[x] \\
\mathrm{sat}(\epsilon) & = & \epsilon \\
\mathrm{sat}(e^+) & = & e^+
\end{array}$$

Next, we extend the saturating operator for grammars. Recall that a grammar G is a finite mapping from variables to compound expressions. We define $\mathrm{sat}(G)$ as a mapping from the same variables to saturated expressions such that

$$\mathrm{sat}(G)(x) \; = !@(\mathcal{N} \setminus \mathrm{att}(G(x)))\;\; \mathrm{sat}(G(x)).$$

The second step creates attribute-element automata from saturated compound expressions. Formally, an *attribute-element automaton* M is an automaton over $\{N[x], @N[x], !@N \mid N \in S, x \in X\}$, where S is a set of name sets and X is a set of variables. We assume that a saturated compound expression is a regular expression whose atoms are $N[x]$, $@N[x]$, or $!@N$. By applying a standard algorithm, we can construct an attibute-element automaton from $\mathrm{sat}(G)(x)$ for every x.

Finally, we present the third step as an operator for rewriting attribute-element automata. Given a set α of attributes, the rewriting operator creates an element automaton from an attribute-element automaton $(Q, q^{\mathrm{init}}, Q^{\mathrm{fin}}, \delta)$. The element automaton borrows Q, q^{init}, and Q^{fin} as the state set, start states, and final states, respectively. However, the transition relation of the element automaton is created by rewriting δ.

First, we preserve transitions for elements. These transitions have labels of the form $N[x]$ and they are captured by δ'_1 (shown below). Second, we rewrite transitions for attributes. For convenience, we use α_N to denote a subset of α such that α_N contains all attributes in α having names in N. For each transition in δ having $@N[x]$ as a label, we introduce a null transition if α_N is non-empty and the value of any attribute in α_N is valid against x with respect to G. These transitions are captured by δ'_2 (shown below). Third, we rewrite transitions for non-existent attribute declarations. For each transition in δ having a label of the form $!@N$, we introduce a null transition if α_N is empty. These null transitions are captured by δ'_3 (shown below). The transition function δ' of the element automaton is defined as the union of δ'_1, δ'_2, and δ'_3. That is,

$$\mathrm{rewrite}_\alpha(Q, q^{\mathrm{init}}, Q^{\mathrm{fin}}, \delta) = (Q, q^{\mathrm{init}}, Q^{\mathrm{fin}}, \delta'),$$

where

$$\begin{array}{rcl}
\alpha_N & = & \{@a[v] \in \alpha \mid a \in N\}, \\
\delta' & = & \delta'_1 \cup \delta'_2 \cup \delta'_3, \\
\delta'_1 & = & (\delta \cap (Q \times \{N[x] \mid N \in S, x \in X\} \times Q) \\
 & & \cup\, (\delta \cap (Q \times Q)), \\
\delta'_2 & = & \{(q_1, q_2) \mid (q_1, @N[x], q_2) \in \delta, \alpha_N \neq \emptyset, \\
 & & \qquad \forall @a[v] \in \alpha_N.(G \vdash v \in x)\}, \\
\delta'_3 & = & \{(q_1, q_2) \mid (q_1, !@N, q_2) \in \delta, \alpha_N = \emptyset\}.
\end{array}$$

We can prove the following theorem, which justifies our validation algorithm.

Theorem 1 *Let* $\alpha = \{@a_1[u_1], @a_2[u_2], \ldots, @a_m[u_m]\}$ *and* $\beta = b_1[v_1]b_2[v_2]\ldots b_n[v_n]$. *Then,* $G \vdash \langle \alpha, \beta \rangle \in G(x)$ *if and only if for some* $y_j \in X$ $(1 \leq j \leq n)$,

1. $G \vdash v_j \in G(y_j)$,
2. $a_k \neq a_l$ $(k \neq l)$, *and*
3. $b_1[y_1]b_2[y_2]\ldots b_n[y_n]$ *is accepted by an element automaton* $\mathit{rewrite}_\alpha(M)$, *where* M *is an attribute-element automaton constructed from* $G(x)$.

4 Boolean Operations

In this section, we present our algorithms for computing intersections of and differences between attribute-element grammars.

The key technique in our algorithms is *partitioning*. Consider first the following intersection of compound expressions.

$$(@a[x]^+ \mid a[x])\,(@b[x]^+ \mid b[x]) \quad\cap\quad @a[y]^+\,(@b[y]^+ \mid b[y]) \tag{1}$$

How can we calculate this intersection? A naive algorithm would separate constraints on attribute sets and those on element sequences in the same way as Section 3.2

$$\begin{array}{ll} & (@a[x]^+\,@b[x]^+ \;\mid\; @a[x]^+\,b[x] \;\mid\; a[x]\,@b[x]^+ \;\mid\; a[x]\,b[x]) \\ \cap & (@a[y]^+\,@b[y]^+ \;\mid\; @a[y]^+\,b[y]) \end{array}$$

and compute the intersection of every pair of clauses on both sides. As before, such use of "distributive laws" makes the algorithm easily blow up. Fortunately, we can avoid it in typical cases. Note that each expression in the formula (1) is the concatenation of two subexpressions, where the left subexpressions on both sides contain the names $@a$ and a and the right subexpressions contain the different names $@b$ and b. In such a case, we can compute intersections of the left subexpressions and of the right subexpressions separately, and concatenate the results:

$$((@a[x]^+ \mid a[x]) \;\cap\; @a[y]^+)\;\;((@b[x]^+ \mid b[x]) \;\cap\; (@b[y]^+ \mid b[y]))$$

The intuition behind why this works is that each "partitioned" expression can be regarded as cross products, and therefore the intersection of the whole expressions can be done by intersecting each corresponding pair of subexpressions. Note also that no subexpression is duplicated by this partitioning process. Therefore the algorithm proceeds linearly in the size of the inputs as long as partitioning can be applied. This idea of splitting expressions into orthogonal parts was inspired by Vouillon's unpublished work on shuffle expressions [27]. We will discuss the difference of our work from his in Section 5.

In the rest of this section, we first describe a preprocessing procedure called "name set normalization." After this, we present definitions for partitioning, our intersection and difference algorithms, and several implementation techniques.

4.1 Name set normalization

Our boolean algorithms take two grammars and first pass these to the *name set normalization* (we simply say "normalization" from now on) phase before the main algorithm. The goal of normalization is to transform the given grammars to equivalent ones such that all name sets appearing in them are pair-wise either equal or disjoint. The reason for using normalization is to simplify our boolean algorithms. For example, consider the following

$$@N_1[x]^+\,@N_2[x]^+ \;\cap\; @N_3[y]^+\,@N_4[y]^+.$$

If N_1 and N_2 are respectively equal to N_3 and N_4, computing this intersection is obvious. However, if these are overlapping in a non-trivial way (e.g., $@\{a,b\}[x]^+$ $@\{c,d\}[x]^+ \cap @\{a,c\}[y]^+ \ @\{b,d\}[y]^+$), it will require more work. Normalization releases us from this tedium.

Let $\{N_1, \ldots, N_k\}$ be the set of name sets appearing in the given grammars. (When we are given two grammars, this set includes all the names both in the grammars.) From this, we generate a set of disjoint name sets by the following shred function. (Here, $\uplus$ is the disjoint union.)

$$
\begin{array}{lcl}
\mathrm{shred}(\{N\} \uplus S) & = & \left\{ \begin{array}{ll} \{N\} \cup \mathrm{shred}(S) \setminus \{\emptyset\} & \text{if } N \cap (\bigcup S) = \emptyset \\[1em] \{N \setminus (\bigcup S)\} \cup & \\ \quad \mathrm{shred}(\{N' \cap N \mid N' \in S\}) \cup & \\ \quad \mathrm{shred}(\{N' \setminus N \mid N' \in S\}) \setminus \{\emptyset\} & \text{otherwise} \end{array} \right. \\
\mathrm{shred}(\emptyset) & = & \emptyset
\end{array}
$$

That is, we pick one member N from the input set. If this member is already disjoint with any other member, we continue shredding for the remaining set S. Otherwise, we first divide each name set N' in S into the disjoint sets $N' \cap N$ and $N' \setminus N$. We separately shred all the name sets of the first form and those of the second form. We then combine the results from these and the name set obtained by subtracting all the members in S from N. As an example, this function shreds the set $\{\{a,b\},\{b,c\},\{a,c\}\}$ in the following way.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{shred}(\{\{a,b\},\{b,c\},\{a,c\}\})} \\
\quad = & \{\{a\}\} \cup \mathrm{shred}(\{\{b\},\{a\}\}) \cup \mathrm{shred}(\{\{c\}\}) \\
\quad = & \{\{a\}\} \cup \{\{b\}\} \cup \{\{a\}\} \cup \{\{c\}\} \\
\quad = & \{\{a\},\{b\},\{c\}\}
\end{array}
$$

This function may blow up since the "otherwise" case above uses two recursive calls to shred where the size of the arguments do not necessarily decrease by half. However, this does not seem to happen in practice since the initial name sets are usually mostly disjoint and therefore the case $N \cap (\bigcup S) = \emptyset$ is taken in most of the time.

We can show the following expected properties.

Lemma 1 *Let* $S = \mathrm{shred}(\{N_1, \ldots, N_k\})$.

1. *For all* $N, N' \in S$, *either* $N = N'$ *or* $N \cap N' = \emptyset$.
2. *For all* N_i, *there is uniquely* $S' \subseteq S$ *such that* $N_i = \bigcup S'$.

Having computed a shredded set $S = \mathrm{shred}(\{N_1, \ldots, N_k\})$, we next replace every occurrence of the form $N[x]$ or $@N[x]^+$ with an expression that contains only name sets in S. We obtain such an expression by using the following norm function.

$$
\begin{array}{lcl}
\mathrm{norm}_S(\emptyset[x]) & = & \emptyset \\
\mathrm{norm}_S((N \uplus N')[x]) & = & N[x] \mid \mathrm{norm}_S(N'[x]) \quad (N \in S) \\
\\
\mathrm{norm}_S(@\emptyset[x]^+) & = & \emptyset \\
\mathrm{norm}_S(@(N \uplus N')[x]^+) & = & \\
\multicolumn{3}{r}{@N[x]^+ \mid ((\epsilon \mid @N[x]^+)\, \mathrm{norm}_S(@N'[x]^+)) \quad (N \in S)}
\end{array}
$$

In the special case that a given $@N[x]^+$ is unioned with ϵ, we can transform it to a somewhat simpler form as follows.

$$
\begin{array}{lcl}
\mathrm{norm}'_S(@\emptyset[x]^+ \mid \epsilon) & = & \epsilon \\
\mathrm{norm}'_S(@(N \uplus N')[x]^+ \mid \epsilon) & = & \\
\multicolumn{3}{r}{(@N[x]^+ \mid \epsilon)\, \mathrm{norm}'_S(@N'[x]^+ \mid \epsilon) \quad (N \in S)}
\end{array}
$$

This specialized rule is important in practice. In our observation, $@N[x]^+$ almost always appears in the form $@N[x]^+ \mid \epsilon$ since the user typically writes zero or more repetitions rather than one or more. Moreover, the straightforward form of concatenations yielded by the specialized rule gives more opportunities to the partitioning technique, compared to the complex form yielded by the general rule, where unions and concatenations are nested each other.

Lemma 2 *Let G be a grammar on X and G' be the normalization of G. Then, $G \vdash v \in G(x)$ iff $G' \vdash v \in G'(x)$ for all v and $x \in X$.*

4.2 Partitioning

In our formalization, it is often convenient to view a nested concatenation of expressions as a flat concatenation and ignore empty sequences in such an expression (e.g., view $(c_1\,(c_2\,\epsilon))\,c_3$ as $c_1\,c_2\,c_3$). In addition, we would like to treat expressions to be "partially commutative," that is, concatenated c_1 and c_2 can be exchanged if one of them is element-free. For example, the expression $@a[x]^+\,(@b[x]^+ \mid b[x])$ is equal to $(@b[x]^+ \mid b[x])\,@a[x]^+$. On the other hand, $(@a[x]^+ \mid a[x])\,(@b[x]^+ \mid b[x])$ is *not* equal to $(@b[x]^+ \mid b[x])\,(@a[x]^+ \mid a[x])$ since, this time, $a[x]$ prevents such an exchange. Formally, we identify expressions up to the relation $\equiv$ defined as follows: $\equiv$ is the smallest congruence relation including the following.

$$
\begin{array}{rcll}
c_1\,c_2 & \equiv & c_2\,c_1 & \text{if } \mathrm{elm}(c_1) = \emptyset \\
c_1\,(c_2\,c_3) & \equiv & (c_1\,c_2)\,c_3 & \\
c\,\epsilon & \equiv & c &
\end{array}
$$

In implementation, it is desirable to represent equal expressions (in terms of $\equiv$) by an identical data structure. For example, we may represent an expression by a pair of a bag of expressions and a list of expressions (e.g., represent $(a[x]\,((@b[x]^+\,c[x])\,\epsilon))$

$@d[x]^+$ by the pair of the bag of $@b[x]^+$ and $@d[x]^+$ and the list of $a[x]$ and $c[x]$).

Now, $(c'_1, c''_1), \ldots, (c'_k, c''_k)$ is a *partition* of $c_1, \ldots, c_k$ if

$$\begin{array}{ll} c_i = c'_i \, c''_i & \text{for all } i \\ (\bigcup_i \operatorname{att}(c'_i)) \cap (\bigcup_i \operatorname{att}(c''_i)) = \emptyset & \\ (\bigcup_i \operatorname{elm}(c'_i)) \cap (\bigcup_i \operatorname{elm}(c''_i)) = \emptyset. & \end{array}$$

That is, each c_i can be split into two subexpressions such that the names contained in all the first subexpressions are disjoint with those contained in all the second subexpressions. We will use partition of 2 expressions ($k = 2$) in the intersection algorithm and that of an arbitrary number of expressions in the difference. The partition is said *proper* when $0 < \operatorname{width}(c'_i) < \operatorname{width}(c_i)$ for some i. Here, *width* counts the number of expressions that are concatenated at the top level (except ϵ). That is,

$$\begin{array}{lcll} \operatorname{width}(\epsilon) & = & 0 & \\ \operatorname{width}(c_1 \, c_2) & = & \operatorname{width}(c_1) + \operatorname{width}(c_2) & \\ \operatorname{width}(c) & = & 1 & \text{if } c \neq \epsilon \text{ and } c \neq c_1 \, c_2 \end{array}$$

(Note that $\equiv$ preserves *width*.) This properness will be used for ensuring the boolean algorithms to make a progress.

4.3 Intersection

Our intersection algorithm is based on product construction. Let grammars F on X and G on Y be given. Assume also that normalization has already been performed *simultaneously* on F and G. (That is, no pair of a name in F and another in G is overlapping.) Now, the intersection of F and G is to compute a new grammar H on $X \times Y$ that satisfies

$$H(\langle x, y \rangle) = \operatorname{inter}(F(x), G(y))$$

for all $x \in X$ and $y \in Y$.

The function inter computes an intersection of compound expressions. It works roughly in the following way. We proceed the computation by progressively decomposing the given compound expressions. At some point, they become attribute-free. Then, we convert the expressions to element automata (defined in Section 3.1), compute an intersection by using a variant of the standard automata-based algorithm, and convert back the result to an expression.

Formally, inter is defined as follows.

$$
\begin{array}{lllll}
1) & \mathrm{inter}(e, f) & = & \mathrm{inter}^{\mathrm{reg}}(e, f) & \\
2) & \mathrm{inter}(e, @N[y]^{+}) & = & \emptyset & \\
3) & \mathrm{inter}(@N[x]^{+}, f) & = & \emptyset & \\
4) & \mathrm{inter}(@N[x]^{+}, @N'[y]^{+}) & = & \emptyset & (N \neq N') \\
5) & \mathrm{inter}(@N[x]^{+}, @N[y]^{+}) & = & @N[\langle x, y\rangle]^{+} & \\
6) & \mathrm{inter}(c, d) & = & \mathrm{inter}(c_1, d_1)\, \mathrm{inter}(c_2, d_2) & \\
 & & & \multicolumn{2}{r}{\text{if } (c_1, c_2), (d_1, d_2) \text{ is a proper partition of } c, d} \\
7) & \mathrm{inter}(c_1\, (c_2 \mid c_3)\, c_4, d) & = & \mathrm{inter}(c_1\, c_2\, c_4, d) \mid \mathrm{inter}(c_1\, c_3\, c_4, d) & \\
8) & \mathrm{inter}(c, d_1\, (d_2 \mid d_3)\, d_4) & = & \mathrm{inter}(c, d_1\, d_2\, d_4) \mid \mathrm{inter}(c, d_1\, d_3\, d_4) &
\end{array}
$$

The base cases are handled by rules 1 through 5, where each of the arguments is either an element expression (as indicated by the metavariables e or f) or an attribute repetition of the form $@N[x]^{+}$. In rule 1, where both arguments are element expressions, we pass them to another intersection function $\mathrm{inter}^{\mathrm{reg}}$ specialized to element expressions. This function will be explained below. Rules 2, 3, and 4 return $\emptyset$ since the argument expressions obviously denote disjoint sets. (Note that, in rule 4, normalization ensures that N and N' are disjoint.) When both arguments are attribute repetitions with the same name set N, rule 5 yields the same form where the content is the intersection of their contents x and y. The inductive cases are handled by rules 6 through 8. Rule 6 applies the partitioning technique already explained. Rules 7 and 8 simply expand one union form appearing in the argument expressions.

Although it may not be obvious at first sight, the presented rules cover all the cases. To see this, first let us view each given expression as a sequence of the form $c_1 \ldots c_k$ where each c_i is either an attribute repetition $@N[x]^{+}$, a union $c_1 \mid c_2$, or an element expression e. Notice that it must be that either (1) one of the two given expressions contains at least one union or (2) none of them has a union. The first case is handled by rules 6, 7 and 8. The actual algorithm applies rule 6 as often as possible, but rules 7 and 8 can always serve as fall backs. (One might think that cases like $\epsilon\,(c_1 \mid c_2)\,\epsilon$ are not handled. However, since, as stated in the previous subsection, we identify expressions up to associativity, partial commutativity, and neutrality of ϵ, such cases are already handled by the aforesaid rules.) In the second case, both of the given expressions are sequences consisting of attribute repetitions and element expressions. The case that both sequences have length zero or one is handled by rule 1 through 5. The remaining case is therefore that either sequence has length two or more. By recalling that attributes are normalized, we can always find a proper partition for such expressions; therefore this case is handled by rule 6.

The intersection function $\mathrm{inter}^{\mathrm{reg}}$ for element expressions performs the following: (1) construct element automata M_1 and M_2 from element expressions e_1 and e_2, (2) compute the "product automaton" M from M_1 and M_2, and (3) convert M back to an element expression e. Since well-known conversion algorithms between automata and regular expressions can directly be used for the case of element automata and element expressions (as described in Section 3.1), we use them for (1) and (3) parts of the $\text{inter}^{\text{reg}}$ function.

The product construction for element automata (used for the (2) part of $\text{inter}^{\text{reg}}$) is slightly different from the standard one. Usually, product construction generates, from two transitions with the same label in the input automata, a new transition with that label in the output automaton. In our case, we generate, from a transition with label $N[x]$ and another with label $N[y]$, a new transition with label $N[\langle x, y\rangle]$. Formally, given two element automata $M_i = (Q_i, q_i^{\text{init}}, Q_i^{\text{fin}}, \delta_i)$ on $\{N[x] \mid N \in S, x \in X_i\}$ $(i = 1, 2)$, the *product* of M_1 and M_2 is an automaton $(Q_1 \times Q_2, \langle q_1^{\text{init}}, q_2^{\text{init}}\rangle, Q_1^{\text{fin}} \times Q_2^{\text{fin}}, \delta)$ on $\{N[\langle x_1, x_2\rangle] \mid N \in S, x_1 \in X_1, x_2 \in X_2\}$ where

$$\delta \quad = \quad \{(\langle q_1, q_2\rangle, N[\langle x_1, x_2\rangle], \langle q_1', q_2'\rangle) \mid (q_i, N[x_i], q_i') \in \delta_i \text{ for } i = 1, 2\}.$$

(Note that we use here the assumption that the name sets of *elements* in the given grammars have been normalized.)

We can prove the following expected property for our intersection algorithm. The proof is presented in Appendix A.

Theorem 2 *Let* $H(\langle x, y\rangle) = \text{inter}(F(x), G(y))$. *Then,* $\text{inter}(c, d) = b$ *implies that* $H \vdash v \in b$ *iff* $F \vdash v \in c$ *and* $G \vdash v \in d$.

4.4 Difference

The difference algorithm is similar to the intersection algorithm since it uses partitioning, but it is somewhat more complicated because of the need to apply subset construction at the same time. To see this, consider the following difference.

$$@a[x]^+\, @b[x]^+ \quad \setminus \quad (@a[y]^+\, @b[y]^+ \mid @a[z]^+\, @b[z]^+)$$

First of all, note that each of the left expression and the right expressions under the union can be partitioned to the one with $@a$ and the one with $@b$; these components can be regarded as orthogonal. We proceed the difference by first subtracting $@a[y]^+\, @b[y]^+$ from $@a[x]^+\, @b[x]^+$. This yields

$$(@a[x]^+ \setminus @a[y]^+)\, @b[x]^+ \;\mid\; @a[x]^+\, (@b[x]^+ \setminus @b[y]^+).$$

That is, we obtain the union of two expressions, one resulting from subtracting the first component and the other resulting from subtracting the second. This can be understood by observing that "a value $@a[v]@b[w]$ being not in $@a[y]^+@b[y]^+$" means "either $@a[v]$ being not in $@a[y]^+$ or $@b[w]$ being not in $@b[y]^+$." Now, back to the original difference calculation, we next subtract the second clause $@a[z]^+\, @b[z]^+$ from the above result. Performing a similar subtraction, we obtain:

$$\begin{array}{lll} & (@a[x]^+ \setminus (@a[y]^+ \mid @a[z]^+)) & @b[x]^+ \\ \mid & (@a[x]^+ \setminus @a[y]^+) & (@b[x]^+ \setminus @b[z]^+) \\ \mid & (@a[x]^+ \setminus @a[z]^+) & (@b[x]^+ \setminus @b[y]^+) \\ \mid & @a[x]^+ & (@b[x]^+ \setminus (@b[y]^+ \mid @b[z]^+)) \end{array}$$

$$
\begin{array}{llcll}
1) & \mathrm{diff}(e,\{f_1,\ldots,f_k\}) & = & \mathrm{diff}^{\mathrm{reg}}(e, f_1 \mid \ldots \mid f_k) & \\
2) & \mathrm{diff}(e,\{@N[y]^+\} \uplus D) & = & \mathrm{diff}(e,D) & \\
3) & \mathrm{diff}(@N[x]^+,\{f\} \uplus D) & = & \mathrm{diff}(@N[x]^+,D) & \\
4) & \mathrm{diff}(@N[x]^+,\{@N'[y]^+\} \uplus D) & = & \mathrm{diff}(@N[x]^+,D) & (N \neq N') \\
5) & \mathrm{diff}(@a[x]^+,\{@a[y_1]^+,\ldots,@a[y_k]^+\}) & = & @a[\langle x,\{y_1,\ldots,y_k\}\rangle]^+ & \\
6) & \mathrm{diff}(@N[\mathbf{any}]^+,\{@N[\mathbf{any}]^+\}) & = & \emptyset & \\
7) & \mathrm{diff}(c,\{d_1,\ldots,d_k\}) & = & \big|_{I\subseteq\{1,\ldots,k\}} \; \begin{array}{l}\mathrm{diff}(c',\{d'_i \mid i \in I\}) \\ \mathrm{diff}(c'',\{d''_i \mid i \in \{1,\ldots,k\} \setminus I\})\end{array} & \\
 & \multicolumn{4}{r}{\text{if } (c',c''),(d'_1,d''_1),\ldots,(d'_k,d''_k) \text{ is a proper partition of } c,d_1,\ldots,d_k} \\
8) & \mathrm{diff}(c_1\,(c_2 \mid c_3)\,c_4,D) & = & \mathrm{diff}(c_1\,c_2\,c_4,D) \mid \mathrm{diff}(c_1\,c_3\,c_4,D) & \\
9) & \mathrm{diff}(c,\{d_1\,(d_2 \mid d_3)\,d_4\} \uplus D) & = & \mathrm{diff}(c,\{d_1\,d_2\,d_4, d_1\,d_3\,d_4\} \uplus D) &
\end{array}
$$

Figure 1: Difference algorithm

Thus, the original goal of difference has reduced to the combination of the subgoals of difference. Let us consider only the first difference $(@a[x]^+ \backslash (@a[y]^+ \mid @a[z]^+))$ since the other are similar. Since all expressions appearing here are attribute repetitions with $@a$, the result is obviously an attribute repetition with $@a$. What is less obvious is that its content is the difference between the variable x and the "union" of the variables y and z. In general, given two grammars, we have to compute the difference between a variable from one grammar and a *set* of variables from the other grammar. This is why the algorithm involves subset construction.

Formally, let grammars F on X and G on Y be given and have been normalized simultaneously, as before. The difference between F and G is to compute a new grammar H on $X \times \mathcal{P}(Y)$ that satisfies

$$H(\langle x, Z\rangle) = \mathrm{diff}(F(x), \{G(y) \mid y \in Z\})$$

for all $x \in X$ and $Z \subseteq Y$. The function diff takes a compound expression c and a set of compound expressions d_i and returns a difference between c and the union of d_i's. The definition of this function is presented in Figure 1. (Here, D ranges over sets of compound expressions and $\uplus$ is the disjoint union.) The base cases are handled by rules 1 through 6. As before, when all the arguments are element expressions, rule 1 passes them to the difference function $\mathrm{diff}^{\mathrm{reg}}$ (explained below). Rules 2, 3, and 4 remove, from the set in the second argument, an expression that is disjoint with the first argument. The remaining base cases are therefore that all the expressions in the arguments are attribute repetitions with the same name set. Recall the restriction on attribute repetitions described in Section 2.2: any expression of the form $@N[x]^+$ must satisfy that either (1) N is singleton or (2) x is **any**. Rule 5 handles the first case, returning an attribute repetition with that singleton name set and an appropriate difference form between variables. Rule 6 handles the second case, returning the empty set expression.

The inductive cases are handled by rule 7 through 9. Rule 8 and 9 expand one union form in the argument expressions. Rule 7 is applied when all the arguments can be partitioned altogether. The complex formula involving "for all subsets $I \subseteq \{1, \ldots, k\}$" on the right hand side is a generalization of the discussion made in the beginning of this section. This "subsetting" technique has repeatedly been used in the literature. Interested readers are referred to [18, 17, 12].

The function $\mathrm{diff}^{\mathrm{reg}}$ is analogous to the function $\mathrm{inter}^{\mathrm{reg}}$ already shown: it constructs element automata M_1 and M_2 from element expressions e_1 and e_2, then computes the "difference automaton" M from M_1 and M_2, and finally converts M back to an element expression e. The construction of difference automata uses both product and subset construction. Given two element automata $M_i = (Q_i, q_i^{\mathrm{init}}, Q_i^{\mathrm{fin}}, \delta_i)$ on $\{N[x] \mid N \in S, x \in X_i\}$ $(i = 1, 2)$, the *difference* of M_1 and M_2 is an automaton $(Q, q^{\mathrm{init}}, Q^{\mathrm{fin}}, \delta)$ on $\{N[\langle x_1, Y_2\rangle] \mid N \in S, x_1 \in X_1, Y_2 \subseteq X_2\}$ where:

$$
\begin{array}{lcl}
Q & = & Q_1 \times \mathcal{P}(Q_2) \\
q^{\mathrm{init}} & = & \langle I_1, \{I_2\}\rangle \\
Q^{\mathrm{fin}} & = & \{\langle q_1, P\rangle \mid q_1 \in F_1 \text{ and } P \subseteq Q_2 \text{ and } P \cap F_2 \neq \emptyset\} \\
\delta & = & \\
\multicolumn{3}{l}{\{\ (\langle q_1, \{p_1, \ldots, p_k\}\rangle, N[\langle x_1, \{y_1, \ldots, y_k\}\rangle], \langle q_1', \{p_1', \ldots, p_k'\}\rangle) \mid} \\
\multicolumn{3}{l}{\quad (q_1, N[x_1], q_1') \in \delta_1 \text{ and } \{(p_1, N[y_1], q_1'), \ldots, (p_k, N[y_k], q_k')\} \subseteq \delta_2\}}
\end{array}
$$

We can prove the following property expected for the difference algorithm.

Theorem 3 *Let $H(\langle x, Z\rangle) = \mathrm{diff}(F(x), \{G(y) \mid y \in Z\})$. Then $\mathrm{diff}(c, D) = b$ implies that $H \vdash v \in b$ iff $F \vdash v \in c$ and $G \vdash v \notin d$ for all $d \in D$.*

Traditional formal language theories typically study complementation rather than difference. One might wonder why we do not do the same. The reason is that, even if we consider complementation, its subcomputation needs to calculate differences anyway. For example, suppose that we have an expression $c\,d$ where $N = \mathrm{att}(c)$ and $N' = \mathrm{elm}(c)$ with $N' \cap \mathrm{elm}(d) = \emptyset$. Then, we would compute the complementation $\overline{c\,d}$ by

$$
((@N[\mathbf{any}]^*\ N'[\mathbf{any}]^*) \setminus c)\, d \quad | \quad c\,((@\overline{N}[\mathbf{any}]^*\ \overline{N'}[\mathbf{any}]^*) \setminus d)
$$

which uses differences. (Note that it is wrong to answer $\overline{c}\,d \mid c\,\overline{d}$ since the names contained in $\overline{c}$ may overlap with those in d.)

So far, we have used the restriction that any attribute repetition has either a singleton name set or the **any** content. What if we remove this restriction? Unfortunately, this seems to break closure under difference. For example, consider computing the difference $@N[\mathbf{any}]^+ \setminus @N[x]^+$. The resulting expression should satisfy the following. Each value in it has a set of attributes all with names from N. But *at least* one of them has a content *not* satisfying x. The expression $@N[\overline{x}]^+$ is not a right answer because it requires *all* attributes to have contents not satisfying x. Although this argument is not a proof, it gives a strong feeling that there is no answer.

It may be worth seeking for an alternative restriction. Here is one possibility. The above argument suggests that the expression we want should look like

$$@N[\overline{x}]^{+}@N[x]^{*}.$$

This expression breaks another restriction: for the disjointness of the name sets in concatenated expressions. Our idea is to relax this. We redefine $\mathrm{att}(c)$ so that it takes contents of attributes into account:

$$\begin{array}{lcl}
\mathrm{att}(@N[x]^{+}) & = & N \times \{v \mid G \vdash v \in G(x)\} \\
\mathrm{att}(c_1\, c_2) & = & \mathrm{att}(c_1) \cup \mathrm{att}(c_2) \\
\mathrm{att}(c_1 \mid c_2) & = & \mathrm{att}(c_1) \cup \mathrm{att}(c_2) \\
\mathrm{att}(e) & = & \emptyset
\end{array}$$

We then require, as before, that $\mathrm{att}(c_1) \cap \mathrm{att}(c_2) = \emptyset$ for each concatenation expression $c_1\, c_2$. This alternative seems plausible because, given a value v satisfying $c_1\, c_2$, we can uniquely determine which of c_1 and c_2 each attribute in v belongs to. However, the correctness remains to be an open question. (We would need an addition restriction that requires expressions for attribute contents to be simple enough so that the disjointness becomes trivial. Also, some external mechanism would be necessary for enforcing attribute names to be disjoint since this alternative restriction does not require this and real XML documents must satisfy this.)

4.5 Implementation techniques

A naive implementation of the algorithms presented above would be prohibitively inefficient. For example, the difference algorithm uses a subset construction, which obviously takes exponential time. Although, in the worst case, we cannot avoid this blow up, we can apply known optimization techniques for making the algorithms much quicker for practical inputs. Below, we briefly explain some of the techniques that we used in our implementation. (More discussions can be found in [18, 17].)

4.5.1 Top-down strategy

The intersection algorithm presented above takes grammars F (on X) and G (on Y) and calculates the product of F and G for the whole domain $X \times Y$. However, the actual algorithm takes "start variables" x_0 and y_0 in addition and the useful part of the output grammar is the one that defines the reachable variables from the output start variable $\langle x_0, y_0 \rangle$. Our observation is that such reachable variables are typically much fewer than the whole $X \times Y$. Exploiting this, we construct the output grammar *lazily* from the start variable $\langle x_0, y_0 \rangle$ in a top-down manner. That is, we initialize H to be a grammar just containing the mapping $\langle x_0, y_0 \rangle \mapsto \mathrm{inter}(F(x_0), G(y_0))$. We then take any pair $\langle x, y \rangle \in \mathrm{FV}(H)$ that is not yet in the domain of H, and add the mapping $\langle x, y \rangle \mapsto \mathrm{inter}(F(x), G(y))$. We repeat this until H becomes self-contained. The same technique can be used in the algorithms for product automata, difference grammars, and difference automata.

4.5.2 Sharing

It is quite common that the intersection algorithm encounters, during its computation, a pair of the same expression $c \cap c$. Since the result is trivially c itself whatever c is, we would like to somehow exploit this fact. However, since expressions can be nested and refer to other variables, sameness is a bit complicated. For example, in the following grammars F and G

$$\begin{array}{ll} F(x_1) = a[x_2] & \quad F(x_2) = \epsilon \\ G(y_1) = a[y_2] & \quad G(y_2) = \epsilon \end{array}$$

x_1 and y_1 are the "same," but detecting this requires a bit of work. In addition, even if we can detect it, we need to copy the whole structure reachable from x_1 to the output grammar.

To deal with this issue, we employ one global grammar. All input and output grammars are parts of it, and, furthermore, these grammars can share some variables. We modify our intersection algorithm so that it takes only two start variables, proceeds the computation using the global grammar, and adds a new mapping to it whenever necessary. Now, the intersection of x and x can be done by returning x itself. Similarly, we can modify our difference algorithm in a way that the difference between x and Y with $x \in Y$ results in a variable $x_\emptyset$ that is assigned to $\emptyset$. The same technique can also be used in the product and difference automata algorithms.

4.5.3 Other techniques

Since the function inter (and similarly for diff) often computes the intersection of the same pair of expressions repeatedly, memoizing the previous computations and reusing the results later are quite helpful. Another technique is to use rules that are specialized to some fixed but frequently used expressions, such as **any** and $\emptyset$. For example, we can use the special rules $x \cap \mathbf{any} = x$ and $x \setminus Y = x_\emptyset$ when $\mathbf{any} \in Y$.

5 Related Work

James Clark [8] has designed a different validation algorithm for attribute-element constraints and implemented it in his validators for TREX and RELAX NG. His algorithm is based on derivatives of regular expressions [4, 1] and proceeds by rewriting a compound expression each time it consumes an attribute or element. Meanwhile, our algorithm allows preprocessors to generate element-attribute automata from a given schema. To validate instance documents, validators can use these element-attribute automata rather than the original schema. Thus, our algorithm is expected to make validators more compact and efficient. To confirm this expectation, we are actively implementing such a preprocessor and validator for RELAX NG.

Our study on attribute constraints has a strong relationship to type theories for record values (i.e., finite mappings from labels to values). Early papers presenting type systems for record types do not consider the union operator and therefore no such complication arises as in our case. (A comprehensive survey of classical records can be found in [13].) Buneman and Pierce have investigated record types with the union operator [5]. Their system does not, however, have any mechanism similar to our attribute repetitions or recursion. The work that is closest to ours may be the recent one by Frisch, Castagna, and Benzaken [12]. In their language called CDuce, they have a typing mechanism for XML attributes that has a similar expressiveness to ours. However, they have not dealt with the algorithmic problems that we did in this paper.

In his unpublished work, Vouillon has considered an algorithm for checking the subset relation between shuffle expressions [27]. His strategy of progressively decomposing given expressions to two orthogonal parts made much influence on our boolean algorithms. The difference is that his algorithm is for a subset checking, which answers just yes or no, and therefore does not incur the complication of switching back and forth between the expression representation and the automata representation, which is needed in our boolean algorithms since they have to reconstruct expressions.

6 Future work

In this paper, we have presented our validation and boolean algorithms. We have already implemented them in the XDuce language [16]. For the examples that we have tried, the performance seems quite reasonable. We plan to collect and analyze data obtained from the experiment on the algorithms in the near future. We also feel that we need some theoretical characterization of the algorithms. In particular, our boolean algorithms contain potentials of blow up in many places. Although our implementation techniques presented in Section 4.5 have been sufficient for our examples, one would like to have some more confidence.

Acknowledgments

We would like to express our warmest thanks to James Clark, Kohsuke Kawaguchi, Benjamin Pierce, and anonymous PLAN-X referees for precious comments and suggestions. Haruo Hosoya has been supported by Japan Society for the Promotion of Science while working on this paper.

References

[BHLC99] Tim Bray, Dave Hollander, Andrew Layman, and James Clark. Namespaces in XML. `http://www.w3.org/TR/REC-xml-names`, 1999.

[BP98] Peter Buneman and Benjamin Pierce. Union types for semistructured data. In *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998. Proceedings of the International Database Programming Languages Workshop.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML^{TM}). `http://www.w3.org/XML/`, 2000.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[BS86] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.

[Cla01a] James Clark. The Design of RELAX NG. `http://www.thaiopensource.com/relaxng/design.html`, 2001.

[Cla01b] James Clark. TREX: Tree Regular Expressions for XML. `http://www.thaiopensource.com/trex/`, 2001.

[Cla02] James Clark. `http://www.thaiopensource.com/relaxng/implement.html`, 2002.

[Fal01] David C. Fallside. XML Schema Part 0: Primer, W3C Recommendation. `http://www.w3.org/TR/xmlschema-0/`, 2001.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science*, 2002.

[FFM$^+$01] Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`, 2001.

[HP91] Robert Harper and Benjamin Pierce. A recrd calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando FL*, pages 131–142. ACM, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[HP00] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997 of *Lecture Notes in Computer Science*, pages 226–244, May 2000.

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, January 2001.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, September 2000. Full version under submission to TOPLAS.

[Int86] International Organization for Standardization. *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986.

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. `http://citeseer.nj.nec.com/murata00taxonomy.html`, 2001.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.

[Mur97] Makoto Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*, volume 1293 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, 1997.

[Mur01] Makoto Murata. RELAX (REgular LAnguage description for XML). `http://www.xml.gr.jp/relax/`, 2001.

[OAS02] OASIS. SGML/XML elements versus attributes. `http://xml.coverpages.org/elementsAndAttrs.html`, 2002.

[Toz01] Akihiko Tozawa. Towards static type inference for XSLT. In *Proceedings of ACM Symposium on Document Engineering*, 2001.

[Vou01] Jérôme Vouillon. Interleaving types for XML. Personal communication, 2001.

A Correctness of the Boolean Algorithms

A.1 Element Automata

From the standard automata theory, the following is well-known.

Proposition 1 ([14])

1. *There is an algorithm* compile *that constructs an automaton M from a given regular expression e such that $L(M) = L(e)$.*

2. *There is an algorithm* decompile *that constructs a regular expression e from a given automaton M such that $L(e) = L(M)$.*

Given a grammar G on X, an *element automaton* is an automaton on $S \times X$, where S is the set of name sets and. We present the semantics of an element automaton M w.r.t. G by the relation $G \vdash v \in M$ defined as follows:

$$\frac{a_i \in N_i \qquad G \vdash v_i \in G(x_i) \qquad N_1[x_1] \ldots N_k[x_k] \in L(M)}{G \vdash a_1[v_1] \ldots a_k[v_k] \in M}$$

Then, we can easily show that this semantics satisfies the following properties.

Corollary 1

1. $G \vdash v \in e$ *iff* $G \vdash v \in \text{compile}(e)$.

2. $G \vdash v \in M$ *iff* $G \vdash v \in \text{decompile}(M)$.

A.2 Intersection

Given two element automata $M_i = (Q_i, q_i^{\mathrm{init}}, Q_i^{\mathrm{fin}}, \delta_i)$ on $S \times X_i$ $(i = 1, 2)$, the *product* of M_1 and M_2 is an automaton $(Q_1 \times Q_2, \langle q_1^{\mathrm{init}}, q_2^{\mathrm{init}}\rangle, Q_1^{\mathrm{fin}} \times Q_2^{\mathrm{fin}}, \delta)$ on $S \times (X_1 \times X_2)$ where

$$\delta \quad = \quad \{(\langle q_1, q_2\rangle, N[\langle x_1, x_2\rangle], \langle q_1', q_2'\rangle) \mid (q_i, N[x_i], q_i') \in \delta_i \text{ for } i = 1, 2\}.$$

By using the standard proof technique, we can show the following expected property of product element automata.

Lemma 3 *Let M be the product automaton of M_1 and M_2 as defined above. Let v be any value. Suppose that, for any x, y, w with $|w| < |v|$,*

$$H \vdash w \in H(\langle x, y\rangle) \textit{ iff } F \vdash w \in F(x) \textit{ and } G \vdash w \in G(y).$$

Then, we have

$$H \vdash v \in M \textit{ iff } F \vdash v \in M_1 \textit{ and } G \vdash v \in M_2.$$

Corollary 2 *Let* $\mathrm{inter}^{\mathrm{reg}}(e_1, e_2) = e$. *With the same assumption as Lemma 3, we have*

$$H \vdash v \in e \textit{ iff } F \vdash v \in e_1 \textit{ and } G \vdash v \in e_2.$$

The following technical lemma shows that, if a value can be partitioned by disjoint sets of attributes and elements, then such a partition is unique.

Lemma 4 *Given a value v, if $v = u\,w = u'\,w'$ where* $(\mathrm{elm}(u) \cup \mathrm{elm}(u')) \cap (\mathrm{elm}(w) \cup \mathrm{elm}(w')) = \emptyset$ *and* $(\mathrm{att}(u) \cup \mathrm{att}(u')) \cap (\mathrm{att}(w) \cup \mathrm{att}(w')) = \emptyset$, *then $u = u'$ and $w = w'$.*

Proof: Let $v = \langle \alpha, \beta\rangle$. Also, let $u = \langle \alpha_1, \beta_1\rangle$ and $w = \langle \alpha_2, \beta_2\rangle$; similarly $u' = \langle \alpha_1', \beta_1'\rangle$ and $w' = \langle \alpha_2', \beta_2'\rangle$. Suppose $u \neq u'$. Then, either $\alpha_1 \neq \alpha_1'$ or $\beta_1 \neq \beta_1'$.

- When $\alpha_1 \neq \alpha_1'$, either $\alpha_1 \setminus \alpha_1' \neq \emptyset$ or $\alpha_1' \setminus \alpha_1 \neq \emptyset$. Therefore $\alpha_1 \cap \alpha_2' \neq \emptyset$ or $\alpha_2 \cap \alpha_1' \neq \emptyset$. Either case contradicts the condition $(\mathrm{att}(u) \cup \mathrm{att}(u')) \cap (\mathrm{att}(w) \cup \mathrm{att}(w')) = \emptyset$.
- When $\beta_1 \neq \beta_1'$, either (1) $\beta_1 = \beta_1'\gamma$ and $\beta_2' = \gamma\beta_2$ for some non-empty γ or (2) $\beta_1' = \beta_1\gamma$ and $\beta_2 = \gamma\beta_2'$ for some non-empty γ. Either case contradicts the condition $(\mathrm{elm}(u) \cup \mathrm{elm}(u')) \cap (\mathrm{elm}(w) \cup \mathrm{elm}(w')) = \emptyset$. □

We use the following function weight from compound expressions to integers for induction in the subsequent proofs.

$$\begin{array}{lcl} \mathrm{weight}(c_1\,c_2) & = & \mathrm{weight}(c_1)\mathrm{weight}(c_2) \\ \mathrm{weight}(c_1 \mid c_2) & = & 1 + \mathrm{weight}(c_1) + \mathrm{weight}(c_2) \\ \mathrm{weight}(c) & = & 1 \quad \text{if } c \neq c_1\,c_2 \text{ and } c \neq c_1 \mid c_2 \end{array}$$

The following lemma shows the correctness of the function inter.

Theorem 4 *Let* $H(\langle x, y\rangle) = \mathrm{inter}(F(x), G(y))$. *Then,* $\mathrm{inter}(c, d) = b$ *implies that* $H \vdash v \in b$ *iff* $F \vdash v \in c$ *and* $G \vdash v \in d$.

PROOF: Let $\mathrm{measure}(v, c, d) = (|v|, \mathrm{weight}(c)+\mathrm{weight}(d), \mathrm{width}(c)+\mathrm{width}(d))$. By induction on the lexicographic order of $\mathrm{measure}(v, c, d)$.

Case: $c = e$ (i.e., $\mathrm{att}(c) = \emptyset$) $\quad d = f \quad b = \mathrm{inter}^{\mathrm{reg}}(e, f)$

The induction hypothesis allows us to use Corollary 2, from which the result follows.

Case: $c = e \quad d = @N[y]^{+} \quad b = \emptyset$ or
$c = @N[x]^{+} \quad d = f \quad b = \emptyset$ or
$c = @N[x]^{+} \quad d = @N'[y]^{+} \quad N \neq N' \quad b = \emptyset$

Trivial.

Case: $c = @N[x]^{+} \quad d = @N[y]^{+} \quad b = @N[\langle x, y\rangle]^{+}$

Clearly, v has the form $@a_1[v_1] \ldots @a_k[v_k]$. By definition, $F \vdash v \in c$ and $G \vdash v \in d$ iff $k \geq 1$ and $a_i \in N$ with $F \vdash v_i \in F(x)$ and $G \vdash v_i \in G(y)$. By induction hypothesis, the last two conditions equal to $H \vdash v_i \in H(\langle x, y\rangle)$. Thus, the result follows.

Case: $b = \mathrm{inter}(c_1, d_1)\,\mathrm{inter}(c_2, d_2)$
$(c_1, c_2), (d_1, d_2)$ is a proper partition of c, d

By the definition of partition, the result suffices to show

$$F \vdash v \in c_1\, c_2 \text{ and } G \vdash v \in d_1\, d_2 \text{ iff } H \vdash v \in \mathrm{inter}(c_1, d_1)\,\mathrm{inter}(c_2, d_2) \tag{2}$$

with $(\mathrm{att}(c_1)\cup\mathrm{att}(d_1))\cap(\mathrm{att}(c_2)\cup\mathrm{att}(d_2)) = \emptyset$ and $(\mathrm{elm}(c_1)\cup\mathrm{elm}(d_1))\cap(\mathrm{elm}(c_2)\cup \mathrm{elm}(d_2)) = \emptyset$. Since the partition is proper, $\mathrm{measure}(v, c_1, d_1) < \mathrm{measure}(v, c, d)$ and $\mathrm{measure}(v, c_2, d_2) < \mathrm{measure}(v, c, d)$. This allows us to apply the induction hypothesis:

$$F \vdash v \in c_1 \text{ and } G \vdash v \in d_1 \text{ iff } H \vdash v \in \mathrm{inter}(c_1, d_1)$$

and similarly for c_2, d_2. From this, the "if" direction of the statement (2) is obvious.

To show the other direction, let v satisfy the left hand side of (2). Then, there are v_1 and v_2 where $v = v_1\, v_2$ with $F \vdash v_1 \in c_1$ and $F \vdash v_2 \in c_2$; also, there are v_1' and v_2' where $v = v_1'\, v_2'$ with $G \vdash v_1' \in d_1$ and $G \vdash v_2' \in d_2$. From Lemma 4, $v_1 = v_1'$ and $v_2 = v_2'$. By the induction hypothesis shown above, we obtain $H \vdash v_1 \in \mathrm{inter}(c_1, d_1)$ and similarly for v_2, from which the result follows.

Case: $c = c_1\,(c_2 \mid c_3)\,c_4 \quad b = \mathrm{inter}(c_1\, c_2\, c_4, d) \mid \mathrm{inter}(c_1\, c_3\, c_4, d)$

Clearly $F \vdash v \in c$ if and only if either $F \vdash v \in c_1\, c_2\, c_4$ or $F \vdash v \in c_1\, c_3\, c_4$. Note $\mathrm{measure}(v, c_1\, c_2\, c_4, d) < \mathrm{measure}(v, c, d)$ and similarly for $c_1\, c_3\, c_4$. The result follows from the induction hypothesis.

Case: $d = d_1\,(d_2 \mid d_3)\,d_4 \quad b = \mathrm{inter}(c, d_1\, d_2\, d_4) \mid \mathrm{inter}(c, d_1\, d_3\, d_4)$

Similar to the previous case. □

A.3 Difference

Given two element automata $M_i = (Q_i, q_i^{\mathrm{init}}, Q_i^{\mathrm{fin}}, \delta_i)$ on $S \times X_i$ $(i = 1, 2)$, the *difference* of M_1 and M_2 is an automaton $(Q, q^{\mathrm{init}}, Q^{\mathrm{fin}}, \delta)$ on $S \times (X_1 \times \mathcal{P}(X_2))$ where:

$$
\begin{array}{lcl}
Q & = & Q_1 \times \mathcal{P}(Q_2) \\
q^{\mathrm{init}} & = & \langle q_1^{\mathrm{init}}, \{q_2^{\mathrm{init}}\}\rangle \\
Q^{\mathrm{fin}} & = & \{\langle q_1, P\rangle \mid q_1 \in Q_1^{\mathrm{fin}} \text{ and } P \subseteq Q_2 \text{ and } P \cap Q_2^{\mathrm{fin}} \neq \emptyset\} \\
\delta & = & \{(\langle q_1, \{p_1, \ldots, p_k\}\rangle, N[\langle x_1, \{y_1, \ldots, y_k\}\rangle], \langle q_1', \{p_1', \ldots, p_k'\}\rangle) \mid \\
 & & (q_1, N[x_1], q_1') \in \delta_1 \text{ and } \{(p_1, N[y_1], q_1'), \ldots, (p_k, N[y_k], q_k')\} \subseteq \delta_2\}
\end{array}
$$

By combining the standard proof technique for product construction and subset construction, we can show the following.

Lemma 5 *Let M be the difference automaton of M_1 and M_2 as defined above. Let v be any value. Suppose that, for any x, y, w with $|w| < |v|$,*

> $H \vdash w \in H(\langle x, Y\rangle)$ *iff* $F \vdash w \in F(x)$ *and* $G \vdash w \notin G(y)$ *for all* $y \in Y$.

Then, we have

> $H \vdash v \in M$ *iff* $F \vdash v \in M_1$ *and* $G \vdash v \notin M_2$.

Corollary 3 *Let* $\mathrm{diff}^{\mathrm{reg}}(e_1, e_2) = e$. *With the same assumption as Lemma 5, we have*

> $H \vdash v \in e$ *iff* $F \vdash v \in e_1$ *and* $G \vdash v \notin e_2$.

Theorem 5 *Let* $H(\langle x, Z\rangle) = \mathrm{diff}(F(x), \{G(y) \mid y \in Z\})$. *Then* $\mathrm{diff}(c, D) = b$ *implies that* $H \vdash v \in b$ *iff* $F \vdash v \in c$ *and* $G \vdash v \notin d$ *for all* $d \in D$.

Proof: Let $\mathrm{measure}(v, c, D)$ be

$$\left(|v|, \mathrm{weight}(c) + \sum_{d \in D} \mathrm{weight}(d), \mathrm{width}(c) + \sum_{d \in D} \mathrm{width}(d)\right).$$

By induction on the lexicographic order of $\mathrm{measure}(v, c, D)$.

Case: $c = e$ $\quad D = \{f_1, \ldots, f_k\}$ $\quad b = \mathrm{diff}^{\mathrm{reg}}(e, f_1 \mid \ldots \mid f_k)$

The induction hypothesis allows us to use Corollary 3, from which the result follows.

Case: $c = e$ $\quad D = \{@N[y]^+\} \uplus D'$ $\quad b = \mathrm{diff}(e, D')$

Since e and $@N[y]^+$ are obviously disjoint, the result follows by the induction hypothesis.

Case: $c = @N[x]^+$ $\quad D = \{f\} \uplus D'$ or $D = \{@N'[y]^+\} \uplus D'$
$b = \mathrm{diff}(@N[y]^+, D')$

Similar to the previous case.

Case: $c = @N[\mathbf{any}]^+ \quad D = \{@N[\mathbf{any}]^+\} \quad b = \emptyset$

Trivial.

Case: $c = @a[x]^+ \quad D = \{@a[y_1]^+, \ldots, @a[y_k]^+\}$
$b = @a[\langle x, \{y_1, \ldots, y_k\}\rangle]^+$

By definition, $F \vdash v \in c$ and $G \vdash v \notin @a[y_i]^+$ for all $i = 1, \ldots, k$ if and only if $l \geq 1$ and $v = @a[v_1] \ldots @a[v_l]$ with $F \vdash v_j \in F(x)$ and $G \vdash v_j \notin G(y_i)$ for all $i = 1, \ldots, k$ and $j = 1, \ldots, l$. By the induction hypothesis, this is equivalent to saying $H \vdash v \in b$.

Case: $D = \{d_1, \ldots, d_k\}$
$b = |_{I \subseteq \{1,\ldots,k\}} \mathrm{diff}(c', \{d'_i \mid i \in I\})\, \mathrm{diff}(c'', \{d''_i \mid i \in \{1, \ldots, k\} \setminus I\})$
$(c', c''), (d'_1, d''_1), \ldots, (d'_k, d''_k)$ is a proper partition of $c, d_1, \ldots, d_k$

By the definition of partition, we have $c = c'\, c''$ and $d_i = d'_i\, d''_i$ with $(\mathrm{att}(c') \cup \bigcup_i \mathrm{att}(d'_i)) \cap (\mathrm{att}(c'') \cup \bigcup_i \mathrm{att}(d''_i)) = \emptyset$ and $(\mathrm{elm}(c') \cup \bigcup_i \mathrm{elm}(d'_i)) \cap (\mathrm{elm}(c'') \cup \bigcup_i \mathrm{elm}(d''_i)) = \emptyset$. We first show the "only if" direction. Let $H \vdash v \in b$. Then, there are v' and v'' such that $H \vdash v' \in \mathrm{diff}(c', \{d'_i \mid i \in I\})$ and $H \vdash v'' \in \mathrm{diff}(c'', \{d''_i \mid i \in \{1, \ldots, k\} \setminus I\})$. By the induction hypothesis, we have

$$F \vdash v' \in c' \text{ and } G \vdash v' \notin d'_i \text{ for all } i \in I$$

and

$$F \vdash v'' \in c'' \text{ and } G \vdash v'' \notin d''_i \text{ for all } i \in \{1, \ldots, k\} \setminus I.$$

These imply that $F \vdash v \in c$ and $G \vdash v \notin d_i$ for all $i \in \{1, \ldots, k\}$.

We next show the "if" direction. Let $F \vdash v \in c$ and $G \vdash v \notin d_i$ for all $i \in \{1, \ldots, k\}$. Then, there are v' and v'' such that $v = v'\, v''$ with $F \vdash v' \in c'$ and $G \vdash v'' \in c''$. From Lemma 4, v' and v'' are uniquely determined. Therefore either $G \vdash v' \notin d'_i$ or $G \vdash v'' \notin d''_i$. Since this holds for all i, there is $I \subseteq \{1, \ldots, k\}$ such that $G \vdash v' \notin d'_i$ for all $i \in I$ and $G \vdash v'' \notin d''_i$ for all $i \in \{1, \ldots, k\} \setminus I$. By the induction hypothesis,

$$H \vdash v' \in \mathrm{diff}(c', \{d'_i \mid i \in I\})$$

and

$$H \vdash v'' \in \mathrm{diff}(c'', \{d''_i \mid i \in \{1, \ldots, k\} \setminus I\}).$$

The desired result $H \vdash v \in b$ follows from these.

Case: $c = c_1\, (c_2 \,|\, c_3)\, c_4 \quad b = \mathrm{diff}(c_1\, c_2\, c_4, D) \,|\, \mathrm{diff}(c_1\, c_3\, c_4, D)$

The result can be shown straightforwardly by using that

$$F \vdash v \in c_1\, (c_2 \,|\, c_3)\, c_4 \text{ iff } F \vdash v \in c_1\, c_2\, c_4 \text{ or } F \vdash v \in c_1\, c_3\, c_4$$

plus the induction hypothesis.

Case: $D = \{d_1\, (d_2 \,|\, d_3)\, d_4\} \uplus D'$
$b = \mathrm{diff}(c, \{d_1\, d_2\, d_4, d_1\, d_3\, d_4\} \uplus D')$

Similar to the previous case. □