

# Cardinality Constraint Automata: A Core Technology for Efficient XML Schema-aware Parsers

Florian Reuter and Norbert Luttenberger  
Institute of Computer Science and Applied Mathematics  
Christian-Albrechts-Universität zu Kiel  
Christian-Albrechts-Platz 4, D-24098 Kiel, Germany  
(flr|nl)@informatik.uni-kiel.de

## ABSTRACT

This article presents a novel class of Finite State Machines called Cardinality Constraint Automata (CCAs). CCAs are especially suited for the construction of XML Schema-aware, validating XML parsers. Parsers built on top CCAs accept richer semantics for XML Schema's *all*, derivation-by-extension, and minimal/maximal occurrence concepts, and are nevertheless extremely efficient. The paper explains the CCA concept and shows how CCA-based parsers are generated from XML Schema definitions. An illustrating example is given to enhance readability of the paper.

## 1. INTRODUCTION

Many new applications — often running in very small devices like PDAs — make use of semistructured data types. All these applications would profit from a semantically rich XML schema definition language with two properties:

- it should allow to specify the largest family of acceptable XML data, and
- it should be possible to construct very efficient parsers for schema instances.

Unfortunately, today's commonly used XML schema languages have some shortcomings with respect to both requirements. For example, the simple and intuitive structure of the XML encoded address book presented in the next section cannot be expressed by current schema languages for XML, including the W3C-standardized XML Schema language.

This article presents Cardinality Constraint Automata (CCAs) which extend the W3C-standardized XML Schema language, and which allow for the implementation of very efficient, XML Schema-aware, validating parsers. CCAs enhance the XML Schema *all* construct, its derivation-by-extension construct, and its minimal/maximal occurrence cardinality expressiveness. CCAs do this by eliminating many of the special cases in the XML Schema specification. For example, the restriction that particles within an *all* construct must not have a maximal occurrence cardinality of more than one can be eliminated completely. The mentioned language enhancements are enabled by the fact that parser construction is not based on a Tree Grammar, but on a novel class of Finite State Machines which we call CCAs according to their most prominent feature of checking cardinality constraints. The semantic enhancements are not

paid for by increased resource consumption; on the contrary, we are confident that CCA-based parsers can be generated even for very small devices.

### 1.1 Illustrating Example

To illustrate the CCA concept we show a parser for an XML schema describing a simple address book as it may occur in Personal Digital Assistants (PDAs). The address book handles only a limited number of addresses as the PDAs memory capacity is limited. In the given example, the address book may contain up to 5000 address entries. An address entry comprises a mandatory name, an optional first name and up to 4 contact informations. A contact is either a mail address or a phone number. The order by which the entries occur in the address entry should not be significant. Figure 1 shows an instance of the address book which contains the author's name, firstname, mail address and phone number.

The related XML schema as given in fig. 2 is invalid with respect to W3C's XML Schema specification because W3C's XML Schema cannot handle *all* declarations where particles occur with a maximum occurrence cardinality of more than one. Valid with respect to W3C's XML Schema, but problematic for current XML Schema-aware parser implementations is the maximal occurrence of 5000 addresses in the address book. For example, the XML Schema-aware parser XERCES[2] translates this occurrence constraint into a regular expression (`address? address? ...`) comprising 5000 `address?` declarations. This is a burden for the computers memory resources. We are going to show that CCAs allow both wider semantics for the *all* construct and can handle maximum occurrence cardinalities without stressing the memory resources. Another benefit of CCAs is the improved semantic for the derivation-by-extension. This CCA feature is not illustrated because it would make the example too complex.

### 1.2 CCA Concept

An XML schema-aware CCA-based XML parser is built from an XML schema in four steps:

1. Transformation to an abstract XML schema model.
2. Generation of an CCA.
3. Elimination of  $\epsilon$ -Transitions.
4. Event Handler construction.

We elaborate on the first two steps in this section of the paper in order to explain the CCA concept. Steps 3 and 4 are explained in the remainder of the paper.

In the first step, the XML representation of the XML schema declaration is transformed into an abstract XML Schema Model. This step is comparable to the generation of an abstract syntax tree in compiler construction. In the abstract XML Schema Model all references are resolved and redundancies are removed.

The abstract XML Schema Model for the Addressbook complex type starts with informations for the Name, Base, Derivation, Content and Definition Type of the Addressbook:

Name	Addressbook
Base	anyType
Derivation	restriction
Content	(Address, Address, 1, 5000)
Definition Type	sequence

The Name field stores the type's name, the Base field stores the type's base type name, the Derivation field stores the derivation method which is either restriction or extension, the Content field stores a sequence of *Particles* and the Definition Type field stores the type of the Content definition which is either *all*, *sequence* or *choice*. A Particle is a quadrupel which stores the particle's tag name or  $\varepsilon$ , the particle's definition type and the particles minimal and maximal occurrence restrictions.

The remaining types declared in the address book schema of figure 2 are represented in the abstract XML Schema Model as follows:

Name	Address
Base	anyType
Derivation	restriction
Content	(Name, anySimpleType, 1, 1), (Firstname, anySimpleType, 0, 1), ( $\varepsilon$ , Contact, 0, 4)
Definition Type	<i>all</i>
Name	Content
Base	anyType
Derivation	restriction
Content	(Mail, anySimpleType, 1, 1) (Phone, anySimpleType, 1, 1)
Definition Type	<i>choice</i>

The second step in the construction chain is the generation of a CCA from the abstract XML Schema Model. The states of a CCA represent the XML schema types. Each state has an associated occurrence counter which is initialized to 0. Transitions roughly correspond to Particles. A transition may have associated Cardinality Constraint Predicates. A transition is triggered by a begin or an end element event, and is executed only when the transition's occurrence cardinality predicate are met. Every time a transition is traversed occurrence counters are updated depending on whether

- a transition is triggered by a begin element event: Then the occurrence counters of all child states are initialized to 0.
- a transition is triggered by an end element event: Then the occurrence counter of the transitions source state is incremented by one.

Consider for example the CCA in figure 3 which represents our address book schema. State 0 is the start state of the automaton. Upon the begin element event `<Addressbook>` the transition to state 1 is traversed which has no occurrence restrictions. While traversing this transition no occurrence counter of state 2 is initialized to zero because the transition

```
<Addressbook>
<Address>
  <Name>Reuter</Name>
  <Mail>flr@informatik.uni-kiel.de</Mail>
  <Phone>+49 880 7296</Phone>
  <Firstname>Florian</Firstname>
</Address>
</Addressbook>
```

**Figure 1: An example instance of the address book schema.**

was triggered by a begin element event. Within the example this indicates that initially no addresses are stored in the address book.

There are two possible transitions for leaving state 1: an end element event `</Addressbook>` triggers the transition back to state 0, and a begin element event `<Address>` triggers the transition to state 2. These two transitions have Cardinality Constraint Predicates expressing constraints on the occurrence of state 2. The transition from state 1 to state 2, for example, can only be traversed if the Cardinality Constraint Predicate  $o(2) < 5000$  is valid. This asserts that the address book currently contains less than 5000 address entries. From  $o(2) < 5000$  together with  $o(2) := o(2) + 1$  upon the end element event `</Address>` it follows that  $o(2) \leq 5000$  remains valid, i.e. the occurrence constraint of maximal 5000 address entries within the address book remains valid.

State 2 has five outgoing transitions. The Cardinality Constraint Predicates attached to these transitions assure that at most one Name, one First Name and five contacts, consisting of either a mail address or a phone number entry, occur. The Cardinality Constraint Predicate of the transition triggered by the end element event `</Address>` asserts that at least a name is specified ( $o(3) > 0$ ).

Obviously, CCAs check for tree structure validity and occurrence constraints in an integrated manner. This integrated checking allows to enhance the semantics of XML Schema by eliminating the restrictions that for instance have been imposed on the *all* construct and to the derivation concept. Furtheron, the constraint checking can be implemented efficiently.

The remainder of this article is organized as follows. After referring to related work, in section 3 an abstract model for the XML Schema Definition language is presented. Thereafter in section 4 CCAs are introduced formally. The generation of CCAs from the abstract model for XML Schema is presented in section 5. Before the construction of XML Schema-aware parsers is addressed in section 7 some preparatory is done in section 6. Section 6 provides a satisfiability condition which expresses whether a sequence of transitions can be possibly traversed and explains how empty transition can be eliminated by calculating the  $\varepsilon$ -closure over these paths. Finally section 8 presents two predicates which specify the validity of an XML Schema definition entirely.

## 2. RELATED WORK: TREE GRAMMARS

Tree Grammars are the current technique to validate the structure of XML Instances [7]. Tree Grammars express the content of types (non terminals) as regular expressions. Therefore Tree Grammars can express the XML Schema *sequence* and *choice* declarations but have problems with the *all* declaration, because the *all* declaration expresses a set of elements. Some work to integrate the set concept into tree grammars has been made in context of checking the validity

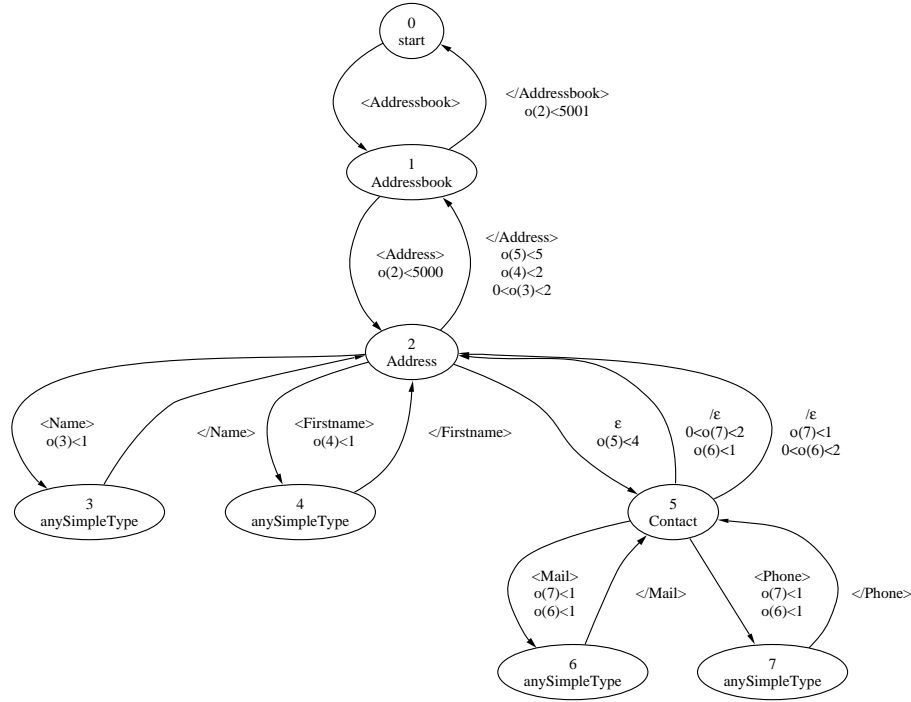


Figure 3: The CCA for the address book schema.

```

<?xml version="1.0" encoding="UTF 8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Addressbook" type="Addressbook" />
  <xs:complexType name="Addressbook">
    <xs:complexContent>
      <xs:restriction base="xs:anyType">
        <xs:sequence>
          <xs:element name="Address" type="Address"
            minOccurs="0" maxOccurs="5000" />
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="Address">
    <xs:complexContent>
      <xs:restriction base="xs:anyType">
        <xs:all>
          <xs:element name="Name" type="xs:anySimpleType"
            minOccurs="1" maxOccurs="1" />
          <xs:element name="Firstname"
            type="xs:anySimpleType" minOccurs="0"
            maxOccurs="1" />
          <xs:group ref="Contact" minOccurs="0" maxOccurs="4"
            />
        </xs:all>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:group name="Contact">
    <xs:choice>
      <xs:element name="Mail" type="xs:anySimpleType" />
      <xs:element name="Phone" type="xs:anySimpleType" />
    </xs:choice>
  </xs:group>
</xs:schema>

```

Figure 2: An XML Schema representing the address book of fig. 1.

of attributes [6, 5].

Besides the *all* construct the other challenging concept for Tree Grammars is the ability of XML Schema to express minimal and maximal occurrences of elements. Regular Expressions and therewith Tree Grammars can only express zero-or-one, minimally one and any number of element occurrences denoted by the *?*, *+* and *\** operators. To express occurrences of minimally 100 and maximal 10000 "e"-elements, a sequence of 100 "e"'s followed by 9900 "e?"'s has to be generated. Obviously, this approach doesn't scale.

The alternative is to replace every element *e* with an occurrence cardinality of more than one by an unconstrained repetition *e\** and perform the constraint check after the parsing process. This is a separating approach. We believe that the integrative approach, the combination of the tree structure check with the check for the occurrence constraints, presented in this article, has advantages compared to the separative approach. The advantages are, as mentioned, the possibility to specify a richer XML Schema semantic.

### 3. XML SCHEMA MODEL

This section presents a syntactical subset of XML Schema which covers all key concepts as defined in the XML Schema Part 1:Structures recommendation[8] except Simple Types and Attributes. This subset, called XML Schema Model, will be used as a substitute for XML Schema in the remaining article for formal argumentation.

An XML Schema Instance is modelled by a tuple

$$(R, T)$$

where *R* is a set of Root Declarations and *T* is a set of Type Declarations.

The set of Root Declarations  $R$  is defined as all tupels

$$(name, type) \in R$$

which contain a  $name \in \text{unicode}^*$  and  $type \in T$  declaration.

The set  $T$  of XML Schema Type Declarations is recursively defined as either a wildcard for any type, a wildcard for any simple type or a complex type or model group declaration.

The anyType type declaration is a wildcard for any XML Schema Type:

$$\text{anyType} \in T$$

The anySimpleType wildcard type declaration stands for the XML Schema Type which does not have children other than values:

$$\text{anySimpleType} \in T$$

Complex Type and Model Group declarations are modelled by the 5 tuple

$$(name, base, derivation, content, defType) \in T$$

where

$name \in \text{unicode}^*$  is the type's name,  
 $base \in T$  is the type's base type,  
 $derivation \in \{\text{extension}, \text{restriction}\}$  is the derivation method, which is either **extension** or **restriction**.  
 $content = \langle p_1, \dots, p_n \rangle$  is a sequence of particles  $p_i, 1 \leq i \leq n$ . The particles  $p_i$  are defined as 4-Tupels

$$p_i := (name, type, minOccurs, maxOccurs)$$

where

$name \in \text{unicode}^*$  is a tag name,  
 $type \in T$  is a type declaration,  
 $minOccurs \in \mathbb{N}$  is a minimal occurrence, and  
 $maxOccurs \in \mathbb{N} \cup \{\infty\}$  is a maximal occurrence,

$defType \in \{\text{all}, \text{sequence}, \text{choice}\}$  specifies the type of the content definition which is either an *all*, *sequence* or *choice* declaration.

The XML Schema Complex Type Declarations can easily be mapped into the presented type system. Group Declarations can be transformed into the 5-tuple by setting the declaration's base type to **anyType** and the derivation method to derivation-by-restriction. The group reference `<group ref="group ref"/>` can be translated into a particle where the tag name is set to the empty word  $\varepsilon$ .

For the sake of simplicity attributes are not modelled. However, they can easily be integrated by extending the above definition with *attributes*.

The original XML Schema specification forces element and group particles within an *all* declaration to have a maximal occurrence cardinality of at most 1. The reason for this is that when Tree Grammars are used for validation, the *all* declaration must be transformed to *choice* and *sequence* declarations. An occurrence cardinality of more than one would make the transformation too complex.

The validation technique based on CCAs has no problems with *all* declaration and occurrence cardinalities of more than 1. For this reason no restrictions are made for particles occurring within *all* declarations.

In fact, only two predicates to express the validity of an XML Schema definition are needed as shown in section 8.

## 4. CARDINALITY CONSTRAINT AUTOMATA

The key concept of the CCA approach is to express cardinality constraints as predicates associated with the transitions of the parser's finite state machine.

A CCA's finite state machine accepts a sequence of begin/end events iff a path through the finite state machine is available for the events (obviously) and all Cardinality Constraint Predicates of the path's transitions evaluate to **true**. Cardinality Constraint Predicates are constructed from terms comprising values generated by the occurrence function  $o(s)$  and the strong and weak merge functions  $\sqcap$  and  $\sqcup$ .

### 4.1 Formal Definition

Formally a CCA is modelled as a 6-tuple

$$(S, E, o, \delta, C, s_0)$$

where

$S$  is a set of states with a function  $type : S \rightarrow T$  which assign every state a type,

$E \subseteq E_S \cup E_E$  is a set of

start element events  $E_S := \{ \langle n \rangle : n \in \text{unicode}^* \}$ ,  
end element events  $E_E := \{ \langle /n \rangle : n \in \text{unicode}^* \}$ ,

$o$  is the initial occurrence function  $o : S \rightarrow \mathbb{N}$ ,

$\delta$  is the transition relation  $\delta \subseteq S \times E \times S \times C$ , and

$C \subseteq P$  is a subset of Cardinality Constraint Predicates.

The set  $P$  of Cardinality Constraint Predicates is defined recursively as

$$\text{true} \in P$$

$$p \in P \implies (i < o(s) < j \wedge p) \in P$$

where  $i, j \in \mathbb{N} \cup \{\infty\}$ .

$s_0 \in S$  is the initial state.

A *configuration* of a CCA is the tuple  $(s, o)$  where  $s \in S$  is the current state and  $o : S \rightarrow \mathbb{N} \cup \{\infty\}$  is the current occurrence function.

A CCA accepts a sequence of events iff for every event a *valid transition* exists. A transition  $(s', e', s'', c')$  is valid iff

1. the source state  $s'$  matches the current state  $s$ ,
2. the event  $e'$  matches the next event in the sequence, and
3. the constraint  $c'$  is valid with respect to the actual occurrence function  $o$ .

A constraint  $p$  is valid when the following equation evaluates to true:

$$\text{isValid}(p, o) :=$$

$$\begin{cases} \text{true} & p = \text{true} \\ \text{false} & p = (i < o(s) < j \wedge p') \in P \wedge \neg(1 < o(s) < j) \\ \text{isValid}(p', o) & p = (i < o(s) < j \wedge p') \in P \wedge 1 < o(s) < j \end{cases}$$

A *valid begin transition* changes the actual configuration to the new configuration  $(s'', o'')$  where  $s''$  is the target state of the transition and  $o''$  is the new occurrence function build

$$\text{by } o''(s) := \begin{cases} o(s) & s \notin \text{adj}_S(s'') \\ 0 & s \in \text{adj}_S(s'') \end{cases} \text{ where } s \in S.$$

A *valid end transition* changes the actual configuration to the new configuration  $(s'', o'')$  where  $s''$  is the target state of the transition and  $o''$  is the new occurrence function build by  $o''(s) := \begin{cases} o(s) & s \neq s'' \\ o(s'') + 1 & s = s'' \end{cases}$  where  $s \in S$ .

The function  $\text{adj}_S(s) := \{s' : (s, e, s', c) \in \delta \wedge e \in E_S\}$  provides all adjacent states of  $s$  which can be reached via a start element event, whereas  $\text{adj}_E(s) := \{s' : (s, e, s', c) \in \delta \wedge e \in E_E\}$  provides all adjacent states reachable via an end element event. The function  $\text{adj}(s) := \text{adj}_S(s) \cup \text{adj}_E(s)$  returns all adjacent states of state  $s$  whereas the function  $\text{out}(s) := \{(s, e, s', c) \in \delta\}$  returns all outgoing incident states of state  $s$ .

## 4.2 Operations for Cardinality Constraint Predicates

This section defines two operations for Cardinality Constraint Predicates: the strong and the weak merge operations. These two operations are used extensively later on to manipulate Cardinality Constraint Predicates. Especially the generation of CCAs out of XML Schema Models relies on these functions.

The strong merge operation merges two Cardinality Constraint Predicates such that the most restrictive upper and lower bounds for each occurrence are taken.

The weak merge operation instead merges two predicates such that the least restrictive bounds are chosen.

For example the Cardinality Constraint Predicates  $(0 < o(1) < 5, (2 < o(2) < 5, \text{true}))$  and  $(1 < o(2) < 4, \text{true})$  strongly merge to  $(0 < o(1) < 5, (2 < o(2) < 4, \text{true}))$ .

The strong merge function  $\sqcap$  is recursively defined as

$$p \sqcap (i < o(s) < j) := \begin{cases} (i < o(s) < j \wedge \text{true}) & \text{when } p = \text{true} \\ (i' < o(s') < j' \wedge p' \sqcap (i < o(s) < j)) & \text{when } p = (i' < o(s') < j' \wedge p') \wedge s \neq s' \\ (\max\{i, i'\} < o(s') < \min\{j, j'\} \wedge p') & \text{when } p = (i' < o(s') < j' \wedge p') \wedge s = s' \end{cases}$$

for a predicate  $p$  and a restriction  $i < o(s) < j$ . The strong merge for two predicates  $p_1$  and  $p_2$  is defined as:

$$p_1 \sqcap p_2 := \begin{cases} p_1 & \text{when } p_2 = \text{true} \\ (p_1 \sqcap (i' < o(s') < j') \sqcap p') & \text{when } p_2 = (i' < o(s') < j' \wedge p') \end{cases}$$

by recursively applying the operation for a predicate and a restriction.

The weak merge function  $\sqcup$  is recursively defined as

$$p \sqcup (i < o(s) < j) := \begin{cases} (i < o(s) < j \wedge \text{true}) & \text{when } p = \text{true} \\ (i' < o(s') < j' \wedge p' \sqcup (i < o(s) < j)) & \text{when } p = (i' < o(s') < j' \wedge p') \wedge s \neq s' \\ (\min\{i, i'\} < o(s') < \max\{j, j'\} \wedge p') & \text{when } p = (i' < o(s') < j' \wedge p') \wedge s = s' \end{cases}$$

for a predicate and a restriction. The weak merge function for two predicates is defined as

$$p_1 \sqcup p_2 := \begin{cases} p_1 & \text{when } p_2 = \text{true} \\ (p_1 \sqcup (i' < o(s') < j') \sqcup p') & \text{when } p_2 = (i' < o(s') < j' \wedge p'). \end{cases}$$

## 4.3 Satisfiability of Cardinality Constraints Predicates

An important criterion of a Cardinality Constraint Predicate is the Satisfiability criteria. Only if restrictions of a Cardinality Constraint Predicate can be satisfied, a corresponding rule has to be included into the parser.

A Cardinality Constraint Predicate  $p$  can be satisfied (denoted as  $\vdash p$ ) if and only if a mapping exists between the occurrence function  $o$  and the natural numbers, such that the predicate evaluates to true:

$$\vdash p : \iff \exists(o' : S \rightarrow \mathbb{N}) : \text{isValid}(p, o')$$

In general the satisfiability problem is NP-hard which is shown by Cook's theorem. However, the satisfiability  $\vdash p$  of a Cardinality Constraint Predicate  $p \in P$  can be easily calculated in linear time by the following function when a state  $s$  only occurs once within an predicate:

$$\vdash p : \iff \begin{cases} \text{true } p = \text{true} \\ \text{false } p = (i < o(s) < j \wedge p') \in P \wedge \neg(j > 0 \wedge i + 1 < j) \\ \vdash p' \quad p = (i < o(s) < j \wedge p') \in P \wedge (j > 0 \wedge i + 1 < j) \end{cases}$$

It is easy to show, if a Cardinality Constraint Predicate is built by successive calls of the weak and the strong merge function every state only occurs once within a predicate.

## 5. GENERATING CARDINALITY CONSTRAINT AUTOMATA

This section defines a function  $\text{GENERATE}(R, T)$  which generates a CCA from a simplified XML Schema Model  $(R, T)$ . The presented  $\text{GENERATE}$  function does not allow recursive XML Schema datatypes, because recursive datatypes need a stack based representation of the occurrence function  $o : S \rightarrow \mathbb{N}$  which is avoided in this paper for clarity and simplicity of the CCA concept.

Figure 5 gives the whole  $\text{GENERATE}$  function in pseudocode notation. A call of the  $\text{GENERATE}(R, T)$  function, where  $(R, T)$  is an instance of the XML Schema Model, results in successive calls of the other  $\text{GENERATE}$ -functions. First the  $\text{GENERATEROOT}$  function is called which generates a new state in the CCA for every root element by calling the  $\text{GENERATETYPE}$  function. The  $\text{GENERATETYPE}$  function either switches to the generation for anyType, anySimpleType or Complex Type. The  $\text{GENERATECOMPLEXTYPE}$  function creates states for all child types and calls, due to the  $\text{GENERATECONTENT}$  function, the appropriate content generating function, which is either  $\text{GENERATEALLCONTENT}$ ,  $\text{GENERATESEQCONTENT}$  or  $\text{GENERATECHOICECONTENT}$ .

The core ideas of the  $\text{GENERATE}$  function and its subfunctions are:

- When a type is derived by extending from a base type, first the content of the base type is generated and then the content of the extending type is added. The content of the extending type is added in a way that the children representing the extending type's content can only be traversed after the children of the base type's content have been traversed.
- Particles are converted to transitions depending on the content type as illustrated in figure 4. Particles of all content are converted to transitions whose Cardinality Constraint Predicates assert the minimal- and maximal occurrence cardinalities. Particles of *sequence*

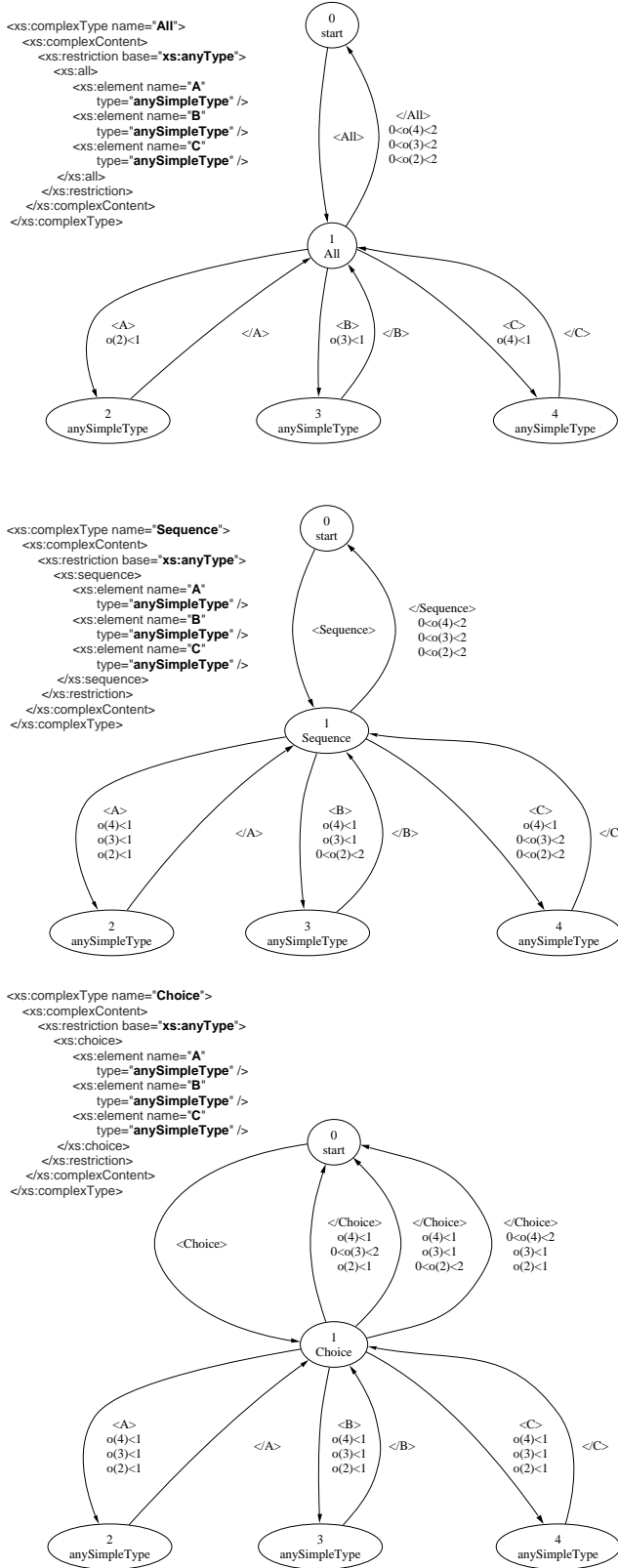


Figure 4: Conversion of different content types, all, sequence and choice, to CCAs.

content are transformed into transitions such that the Cardinality Constraint Predicates assert the order of the predicates and the occurrence cardinality restrictions. Particles of *choice* content are transformed into separate transitions whose Cardinality Constraint Predicates assert beneath the occurrence cardinality restrictions that only one transition can be traversed each time.

## 6. ELIMINATION OF $\epsilon$ TRANSITIONS

This section describes the elimination of  $\epsilon$  transitions which make the choice indeterministic for a parser. The  $\epsilon$  transitions are eliminated by calculating the  $\epsilon$ -closure. Because this operation may exponentially increase the number of states it is important to build the closure only about transitions which could actually be traversed, i.e. can occur in real parser runs. Therefore special paths in CCAs called Event Paths and their important Satisfiability Property are introduced. Event Paths are used to calculate the  $\epsilon$ -closure of CCAs.

### 6.1 Event Paths

A path is defined as a sequence of transitions

$$P = \langle t_1, \dots, t_n \rangle \quad t_i \in \delta$$

with the property that

$$\text{dest}(t_i) = \text{source}(t_{i+1})$$

holds for every  $i \in \{1, \dots, n-1\}$  and that the path is loop free. A path is called loopfree if there do not exist two transitions which have the same source and destination states, regardless of the Cardinality Constraint Predicates and the tags. Formally:

$$\text{loopfree}(P) := \forall t_i, t_j \in P :$$

$$t_i \neq t_j \implies \text{source}(t_i) \neq \text{source}(t_j) \wedge \text{dest}(t_i) \neq \text{dest}(t_j).$$

An path where each transition is satisfiable during the traversal of a CCA is called a Event Path. In other words only Event Paths can occur in real parser runs.

Consider for example the path  $\langle (1, \langle \text{Address} \rangle, 2, o(2) < 5000, \text{true}), (2, \langle / \text{Address} \rangle, 1, o(5) < 5 \wedge o(4) < 2 \wedge 0 < o(3) < 2) \rangle$  from state 1 via state 2 to state 1 in the CCA shown in fig. 3. Although the path is loop-free it is no Event Path because the first transition from state 1 to state 2 resets the occurrence function  $o(3)$  to zero which means that the transition back to state 1 can never be traversed thereafter because of the constraint  $0 < o(3) < 2$ . Concrete this means an empty address  $\langle \text{Address} / \rangle$  is not allowed because minimally a name is required.

The remainder of this section first introduces a predicate second which transforms Event Paths into Cardinality Constraint Predicates which describe the satisfiability condition. The satisfiability property of Event Paths is then defined by using this satisfiability condition.

The satisfiability condition for a predicate  $p$ , a set of states

```

GENERATE(R, T)
  A := newCCA();
  s0 := newStartState(A);
  forall r ∈ R do
    GENERATEROOT(A, s0, r, T);
  return A;

GENERATEROOT(A, s0, r, T)
  state := GENERATETYPE(A, s0, r.name, r.type);
  addBeginTrans(A, s0, r.name, state, true);
  return;

GENERATETYPE(A, parent, tag, type)
  if type = anyType then
    state := newState(A, anyType);
    addEndTrans(A, state, tag, parent, true);
  elif type = anySimpleType then
    state := newState(A, anySimpleType);
    addEndTrans(A, state, tag, parent, true);
  else
    state := GENERATECOMPLEXTYPE(A, parent, tag, type);
  fi
  return state;

GENERATECOMPLEXTYPE(A, parent, tag, type)
  state := newState(A, type);
  if type.derivation=extension then
    eConds := GENERATECONTENT(A, state, type.base, true);
    forall eCond ∈ eConds do
      eCond' :=
        GENERATECONTENT(A, state, type, eCond);
      forall eCond'' ∈ eConds' do
        addEndTrans(A, state, tag, parent, eCond' ∩ eCond'');
      od;
    od;
  else
    eConds := GENERATECONTENT(A, state, type, true);
    forall eCond ∈ eConds do
      addEndTrans(A, state, tag, parent, eCond);
    od;
  fi;
  return state;

GENERATECONTENT(A, state, type, bCond)
  if type.defType=all then
    C := GENALLCONTENT(A, state, type.content, bCond);
  elif type.defType=sequence then
    C := GENSEQCONTENT(A, state, type.content, bCond);
  elif type.defType=choice then
    C := GENCHOICECONTENT(A, state, type.content, bCond);
  fi;
  return C;

GENALLCONTENT(A, state, C, bCond)
  eCond := true;
  forall particle ∈ C do
    child := GENERATETYPE(A, state, particle.tag, particle.type);
    addBeginTrans(A, state, particle.tag, child,
      bCond ∩ (−1 < o(child) < particle.maxOccurs));
    eCond := eCond ∩
      (particle.minOccurs − 1 < o(child) < particle.maxOccurs + 1);
  od;
  return {eCond};

GENSEQCONTENT(A, state, C, bCond)
  eCond := bCond;
  forall particle ∈ C do
    child[particle] :=
      GENERATECONTENT(A, state, particle.tag, particle.type);
    eCond := eCond ∩ (−1 < o(child[particle]) < 1);
  od;
  forall particle ∈ C do
    addBeginTrans(A, state, particle.tag, child[particle],
      eCond ∩ (−1 < o(child[particle]) < particle.maxOccurs));
    eCond := eCond ∩
      (∞ < o(child[particle]) < particle.maxOccurs + 1);
  od;
  return {eCond};

GENCHOICECONTENT(A, state, C, bCond)
  eConds := {};
  forall particle ∈ C do
    child[particle] :=
      GENERATECONTENT(A, state, particle.tag, particle.type);
  od;
  forall particle ∈ C do
    bCond' := bCond;
    eCond' := true;
    forall particle' ∈ C do
      if particle = particle' then
        bCond' := bCond' ∩
          (−1 < o(child[particle]) < particle.maxOccurs);
        eCond' := eCond' ∩
          (particle.minOccurs − 1 < o(child[particle]) < particle.maxOccurs + 1);
      else
        bCond' := bCond' ∩ (−1 < o(child[particle']) < 1);
        eCond' := eCond' ∩ (−1 < o(child[particle']) < 1);
      fi
    od
    addBeginTrans(A, state, particle.tag, child[particle], bCond');
    eConds := eConds ∪ {eCond'};
  od
  return eConds;

```

Figure 5: The pseudocode for the Generate function

$Z \subseteq S$  and a function  $a : S \rightarrow \mathbb{N}$  is defined as

$$\text{scond}(p, Z, a) := \begin{cases} \text{true} & p = \text{true} \\ \text{scond}(p', Z, a) & p = (i < o(s) < j, p') \wedge s \in Z \wedge i < a(s) \wedge a(s) < j \\ (0 < o(s) < 0, \text{true}) & p = (i < o(s) < j, p') \wedge s \in Z \wedge \neg(i < a(s) \wedge a(s) < j) \\ (i - a(s) < o(s) < j - a(s), \text{scond}(p', Z, a)) & p = (i < o(s) < j, p') \wedge s \notin Z. \end{cases}$$

The set  $Z$  of states holds the states of which it is known that the value of the occurrence function for this states is 0. The function  $a$  defines for each state an additive constant.

The above function works as follows. If the passed predicate is **true** then the generated predicate is **true** also. If the passed predicate consists of a constraint  $i < o(s) < j$  and a rest predicate  $p'$  then the satisfiability condition depends on the following cases:

1. The occurrence function  $o(s)$  of the state  $s$  has been set to zero  $s \in Z$  and the predicate  $i < o(s) + a(s) < j \Leftrightarrow i < 0 + a(s) < j \Leftrightarrow i - a(s) < 0 < j - a(s) \Leftrightarrow i < a(s) \wedge a(s) < j$  is satisfiable, then the satisfiability condition for the predicate only depends on the rest  $p'$ .
2. The occurrence function  $o(s)$  of the state  $s$  has been set to zero  $s \in Z$  and the predicate  $i < a(s) \wedge a(s) < j$  is not satisfiable, then the satisfiability condition of the predicate is **false** — denoted by the Cardinality Constraint Predicate  $(0 < o(s) < 0, \text{true})$  which cannot be satisfied and such represents the boolean constant **false**.
3. The value of the occurrence function  $o(s)$  of the state  $s$  is unknown ( $s \notin S$ ) and thus the satisfiability condition  $i < o(s) + a(s) < j \Leftrightarrow i - a(s) < o(s) < j - a(s)$  is concatenated with the satisfiability condition of the rest predicate  $p'$ .

The satisfiability condition of a path  $P$  is defined using the satisfiability condition for predicates as follows:

$$\text{scond}(P, Z, a) := \begin{cases} \text{true} & P = \varepsilon \\ \text{scond}(c, Z \cup \text{adj}_S(d), \text{reset}(a, \text{adj}_S(d))) & \\ \sqcap \text{scond}(P', Z \cup \text{adj}_S(d), \text{reset}(a, \text{adj}_S(d))) & P = (s, e, d, c) \circ P' \wedge e \in E_S \\ \text{scond}(c, Z, \text{inc}(a, \{s\})) \sqcap \text{scond}(P', Z, \text{inc}(a, \{s\})) & P = (s, e, d, c) \circ P' \wedge e \in E_E \end{cases}$$

The above function simply traverses each transition  $(s, e, d, c)$  in the Event Path  $P$  in the way that

1. if the transition is a begin transition  $e \in E_S$  then all  $\text{adj}_S$  children are set to zero according to the semantic of valid begin transitions given in section 4.
2. if the transition is an end transition  $e \in E_E$  the additive constant  $a(s)$  is increased by one according to the semantic of valid end transitions defined in section 4.

The auxiliary functions  $\text{reset}(a, S)$  which sets the additive constant for all states in  $S$  to zero and  $\text{inc}(a, S)$  which increases the additive constant of all states in  $S$  by one are

defined pointwise as follows:

$$\text{reset}(a, S)(s) := \begin{cases} a(s) & s \in S \\ 0 & s \notin S, \end{cases} \quad \text{inc}(a, S)(s) := \begin{cases} a(s) + 1 & s \in S \\ a(s) & s \notin S. \end{cases}$$

The satisfiability condition of an Event Path  $P$  is simply defined as

$$\text{scond}(P) := \text{scond}(P, \emptyset, \bar{0})$$

where the initial value for  $Z$  is the empty set and for  $a$  is the function  $\bar{0}$  which constantly maps all states to 0.

The validity  $\vdash P$  of an Event Path can thus be defined as

$$\vdash P : \iff \vdash \text{scond}(P).$$

## 6.2 $\varepsilon$ -closure

The idea behind the elimination of the  $\varepsilon$  transitions is to calculate for every state the set of Event Paths which end with a non- $\varepsilon$  transition and begin with minimally none  $\varepsilon$  transition. Having these  $\varepsilon$ -paths for each state the next section will show how to implement validating parser.

The  $\varepsilon$ -paths of a state  $s$  are calculated by building the  $\varepsilon$ -closure over all incident outgoing transitions:

$$\varepsilon\text{-paths}(s) := \varepsilon\text{-closure} \left( \bigcup_{t \in \text{out}(s)} \{ \langle t \rangle \} \right).$$

The  $\varepsilon$ -closure of a state  $s$  is the least fixpoint of the function which adds all satisfiable and loopfree Event Paths originating from the state  $s$  and whose last transition is a  $\varepsilon$  transition. The pseudocode for the  $\varepsilon$ -closure is given below:

```

\varepsilon\text{-closure}(\mathcal{P})
  changes:=true;
  while changes do
    changes:=false;
    forall P \in \mathcal{P} do
      if event(last(P))=\varepsilon then
        \mathcal{P}:=\mathcal{P} \setminus \{P\}
        forall t \in \text{out}(\text{dest}(\text{last}(P))) do
          if \vdash P \circ \langle t \rangle \wedge \text{loopfree}(P \circ \langle t \rangle) then
            \mathcal{P}:=\mathcal{P} \cup \{P \circ \langle t \rangle\}
            changes:=true;
          fi
        od
      fi
    od
  fi
od
od
return \mathcal{P}

```

## 7. PARSER IMPLEMENTATION

This section describes how validating, XML Schema-aware XML Parsers can be built on top of CCAs. These validating parsers make use of SAX parsers[1] because of two advantages:

- Pure Automata, and thus also CCAs, cannot check the well-formedness of XML Instances because an additional stack is needed to keep track of the begin and corresponding end element events. SAX parsers check for well-formedness, so parsers built on top of the SAX API can rely on the well-formedness property.
- SAX parsers generate a (well-formed) sequence of begin and end elements which can directly be used to trigger transitions of the CCA from one configuration  $(s, o)$  to another.

In order to integrate an XML Schema-aware, validating XML parser with a SAX parser an event handler has to be implemented. The following pseudocode represents such an SAX event handler. Please note that the SAX API requests different handlers for begin and end element events. The presented pseudocode thus has to be doubled and slightly modified to fit in real SAX implementations:

```

SAX-EVENT-HANDLER(e)
  \mathcal{P}:=\text{event paths generated by } \varepsilon\text{-paths for state } s;
  error:=true;
  forall P \in \mathcal{P} do
    if event(last(P)) = e \wedge \text{scond}(P) then
      forall (s', e', d', c') \in P do
        if e' \in E_S then
          s:=d';
          forall s'' \in \text{adj}_S(d') do o(s''):=0 od;
          error:=false;
        else
          s:=d';
          o(s'):=o(s') + 1;
          error:=false;
        fi
      od
    od
  if error then halt;

```

When the SAX-EVENT-HANDLER is called by the SAX-API with a SAX-Event  $e$ , the handler first gathers the Event Paths generated by the  $\varepsilon$ -paths( $s$ ) function for the current state  $s$  of the current configuration  $(s, o)$ . Usually this step is implemented as a simple table lookup to a table which stores the precalculated Event Paths for every state.

The handler then checks for every Event Path  $P$  whether the current SAX-Event  $e$  matches the last event of the Event Path  $P$  which is a non  $\varepsilon$ -transition as asserted by the  $\varepsilon$ -closure operation. When the events match it is checked whether the cardinality constraints denoted by  $\text{scond}(P)$  are valid with respect to the actual occurrence function  $o$  of the current configuration  $(c, o)$ .

When the events match and the occurrence restrictions are valid with respect to the actual occurrence function, every transition in the Event Path is traversed according to the semantic of CCAs specified in section 4:

- For every begin transition the actual configuration  $(s, o)$  is changed in a way that the destination of the begin transition is the new state and that the occurrence cardinalities of all  $\text{adj}_S$  children are set to zero.
- For every end transition the actual configuration  $(s, o)$  is changed such that the destination of the end transition is the new state and that the occurrence cardinality of the transitions source state is increased by one.

If no Event Path can be chosen then the XML Instance represented by the SAX Events is not valid with respect to the XML Schema represented by the CCA's Event Paths.

The initial configuration of the CCA should be set to

$$(s_0, \bar{0})$$

where  $s_0$  is the start state and  $\bar{0}$  maps every state's occurrence to zero.

## 8. SCHEMA-VALIDITY

The original XML Schema specification defines a huge amount of constraints which express the validity of an XML

Schema Declaration. This is, because the original XML Schema can syntactically express structures which cannot be handled semantically. Examples are the problem of maximal occurrences in *all* constructs of more than one and the derivation-by-extension or restriction which are only valid for a few cases which are syntactically possible. More precisely the derivation-by-extension is only semantically defined for the case of *sequences*. The derivation-by-restriction specification has been exposed to criticism since it can produce inconsistencies [4].

The presented (simplified) XML Schema Syntax and Semantic only needs two validity constraints: A Uniqueness constraint and a constraint for the derivation-by-restriction.

## 8.1 Uniqueness Constraint

An XML Schema-aware parser traverses the first transition which matches the current element event and whose cardinality constraints are satisfied. In order to make this choice deterministic for the parser it has to be asserted that only one transition can be chosen for each event. The following uniqueness predicate formalizes this constraint

$$\forall s \in S \forall P_1, P_2 \in \varepsilon\text{-paths}(s) : \\ P_1 \neq P_2 \implies \neg \text{impl}(\text{scond}(P_1), \text{scond}(P_2))$$

with the  $\text{impl}$  operation defined as

$$\text{impl}(p_1, p_2) := \vdash p_1 \implies \vdash p_2.$$

The  $\text{impl}(p_1, p_2)$  evaluates to true when the satisfiability of predicate  $p_1$  implies the satisfiability of predicate  $p_2$ . The uniqueness predicate can be checked efficiently in polynomial time if the following implementation for the  $\text{impl}$  operation is used:

$$\text{impl}(p_1, p_2) := \begin{cases} \vdash p_2 & p_1 = \text{true} \\ \text{impl}(i < o(s) < j, p_2) \wedge \text{impl}(p, p_2) & p_1 = (i < o(s) < j, p') \end{cases}$$

for two predicates  $p_1, p_2$  and

$$\text{impl}(i < o(s) < j, p) := \begin{cases} \text{true} & p = \text{true} \\ i' \leq i \wedge j \leq j' & p = (i' < o(s') < j', p') \wedge s' = s \\ \text{impl}(i < o(s) < j, p') & p = (i' < o(s') < j', p') \wedge s' \neq s \end{cases}$$

for a predicate  $p$  and a cardinality constraint  $i < o(s) < j$ .

## 8.2 Derivation-By-Restriction Validity

The validity for the derivation-by-restriction construct presented here is the one suggested by Brown et. al. who roughly spoken defined the derivation-by-restriction as "one type is a restriction of another if every instance of the first type is also an instance of the second" [4].

This validity constraint for the derivation-by-restriction is expressed by the predicate

$$\exists \pi \forall (s, e, d, c) \in \text{out}(s_1) \exists (s', e', d', c') \in \text{out}(s_2) : \\ e = e' \wedge d = \pi(d') \wedge \text{impl}(c, \text{relabel}(c', \pi)) \wedge \text{type}(d) = \text{type}(d')$$

where  $\pi : \text{adj}(s_1) \cup \{s_0\} \rightarrow \text{adj}(s_2) \cup \{s_0\}$  is a function from the adjacent states of state  $s_1$  to the adjacent states of state  $s_2$ . The state  $s_0$  is the start state of a CCA  $A$  which serves as the parent state for  $s_1$  as well as for  $s_2$ .

The states  $s_1$  and  $s_2$  can be generated by the two function calls:

$$s_1 = \text{GENERATECOMPLEXTYPE}(A, s_0, \varepsilon, \text{type}) \\ s_2 = \text{GENERATECOMPLEXTYPE}(A, s_0, \varepsilon, \text{type.base}).$$

The auxiliary function  $\text{relabel}$  renames the states in the predicate  $p$  to the states pointed by  $\pi$ :

$$\text{relabel}(p, \pi) := \begin{cases} \text{true} & p = \text{true} \\ (i < o(\pi(s)) < j \wedge p') & p = (i < o(s) < j \wedge p'). \end{cases}$$

The current implementation, which checks the validity constraint of the derivation-by-restriction construct, uses a backtracking algorithm which in worst case has an exponential running time. However, experiences with the prototype implementation show that the running time is short for practical cases.

## 9. OPEN ISSUES

This article raised two open issues which could not be addressed because of space restrictions. The first open issue is the constraint that the XML Schema Declaration must not contain recursive data types, the second is the question how to support the `xsi:type` construct needed for XML Schema's derivation concept. The following hints help to solve these problems:

- To support recursive data types the occurrence function  $o : S \rightarrow \mathbb{N}$  needs to be maintained in a stack based data structure. The `GENERATETYPE` function could then be modified not to create a new state for every XML Schema type but to link to an existing state and refer to the occurrence values on the stack.
- The `xsi:type` construct is best addressed in the parser generation step. Here code has to be inserted which evaluates the `xsi:type` attribute and traverses to the state with the specified type.

## 10. CONCLUSION

We have presented CCAs as a powerful core technology for XML Schema-aware, validating parsers. CCAs allow to construct efficient parsers with a richer semantic compared to the original XML Schema semantic designed for Tree Grammars. CCA-based parsers can especially overcome the problem that Tree Grammar-based parsers have with the *all* declaration, the derivation-by-extension mechanism and the minimal and maximal occurrence cardinality specification.

A prototype implementation of the presented techniques has been built within the Swarms project [3] where efficient parsers are needed for small, memory-constraint devices.

Future versions of the prototype will integrate Attributes and Simple Types. The efforts of implementing a fully W3C XML Schema-conformant parser based on CCAs are currently evaluated.

Future work will focus on the serialization and de-serialization of XML documents to programming language data types.

## 11. REFERENCES

- [1] SAX: Simple API for XML.  
<http://www.saxproject.org/>.

- [2] Schema implementation limitations.  
<http://xml.apache.org/xerces-j/schema.html>.
- [3] SWARMS - Software Architecture for Radio-based Mobile Systems. <http://www.swarms.de>.
- [4] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL: a model for W3C XML schema. *Computer Networks*, 39:681–697, 2002.
- [5] J. Clark.  
An algorithm for RELAX NG validation, February 2002.  
<http://www.thaiopensource.com/relaxng/derivative.html>.
- [6] H. Hosoya and M. Murata. Validation and boolean operations for attribute-element constraints. In *Programming Language Technologies for XML (PLAN-X)*, Oct. 2002.  
<http://www.research.avayalabs.com/user/wadler/planx/planx-eproceed/proceed.html>.
- [7] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.
- [8] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. W3C Recommendation, May 2001.  
<http://www.w3.org/TR/xmlschema-1/>.