Chapter 5
Specification of XML-Documents

I. INTRODUCTION

AIM of the chapter is to give an implementation independent (algebraic) description of data structures, which generalize XML-documents and database tables. Our specification is based on the following 8 generating operations, which are illustrated each by an example.

Empty_t ⟶ Tabment
(Empty table with Empty scheme)
t0 = <></>

El_tab (Value) ⟶ Tabment
(table contains one elementary value)
ta = El_tab(a) = <string>a</string>
It would be also possible to consider a float as an elementary value:
t1 = El_tab(1.234) = <float>1.234</float>

Empty (s: Scheme iff s is a collection scheme) ⟶ Tabment
(empty table of a collection scheme)
t2 = Empty(L(A, L(B))) = <(A, B*)*></(A, B*)*>

Tag0 (n: Name, t: Tabment iff type_n(n) = type_t(t)) ⟶ Tabment
(enclose a table *t* by an additional tag *n)*
t3 = Tag0(A, t1)
= <A><float>1.234</float></A>
= <A>1.234</A>
t4 = Tag0(B, El_tab(2.345)) = <B>2.345</B>

Pair (Tabment, Tabment) ⟶ Tabment
(build a Pair (2-tuple))
t5 = Pair(t3, t4) = <A,B><A>1.234</A>
<B>2.345</B>
</A,B>

Add (t1: Tabment, t2: Tabment iff t2 is of element type of t1 or coll_type_t(t1)=Any) ⟶ Tabment
(Add a table t2, which is of element type of t1, to t1)
Add(Empty(L(A, B)), t5) =
= <(A,B)*><A,B><A>1.234</A><B>2.345</B></A,B>
</(A,B)*>

Alternate_t (t: Tabment, s: Scheme) ⟶ Tabment
(extend the scheme of table t to an alternative)
Alternate(t3, B) = <A | B><A>1.234</A><A | B>

On the base of these generating operations powerful and user-friendly operations can be specified. Stroke for example is an operation, which allows a restructuring of arbitrary XML-documents to another XML-document, only if the target scheme is given.

II. SPECIFICATION OF SCHEMES OF DOCUMENTS

The specification of XML-documents requires a precision of the notion of a scheme of a document. Our algebraic specification uses initial semantic. That means we can represent all elements of a sort by terms. Two terms are equal if and only if the equality can be deduced by the given implications. An operation is allowed to have defining conditions. Such operations are partial. They produce a result only if the corresponding elements of sorts satisfy the defining conditions (for details see H. Reichel, „Initial Computability, Algebraic Specifications, and Partial Algebras“, Akademie Verlag Berlin (Oxford-Press) 1987).

```
def
sorts   Bool, Nat                    // Boolean values and natural numbers
opers   true, false ⟶ Bool
        zero, one ⟶ Nat
        succ (Nat) ⟶ Nat             // successor of a natural number
        (Nat +, * Nat) ⟶ Nat         // addition and multiplication
        (Nat <, >,...Nat) ⟶ Bool     // smaller-relation, ...
        and, or (Bool, Bool) ⟶ Bool
axioms x, y: Nat
        succ(zero) = one
        x + zero = x
        x + succ(y) = succ(x+y)
        ...
end
def
sorts   Coll-sym                     // collection symbols
opers   Set, Bag, List, S1, Any ⟶ Coll-sym
end
```

Symbols for sets, multiset, lists, optional elements (S1) and heterogeneous collections (Any-elements). An optional element is considered as a set with at most one element.

```
def
sorts   Value   // elementary values =Strings+Ints+Floats+ Booleans+Bar
opers   String_v (string) ⟶ Value
        Int_v (Int) ⟶ Value
        Float_v (Float) ⟶ Value
        Bool_v (Bool) ⟶ Value
        Bar ⟶ Value
end
def
sorts   Name                         // names for elementary tags
opers   ZAHL, TEXT,... ⟶ Name
        subject, mark, result, pupil,... ⟶ Name
end
sorts   Scheme
opers   Empty_s ⟶ Scheme                       // Empty scheme
        Inj (Name) ⟶ Scheme                    // each name is a scheme
        Pair_s (Scheme, Scheme) ⟶ Scheme       // 2-tuple of schemes
```

Coll_s (Coll-sym, Scheme) ⟶ Scheme
Alternate_s (Scheme, Scheme) ⟶ Scheme
axioms s, s', s": Scheme
Pair_s(s, Empty_s) = Pair_s(Empty_s, s) = s
Pair_s(Pair_s(s, s'), s") = Pair_s(s, Pair_s(s', s"))
Alternate_s(Alternate_s(s, s'), s") = Alternate_s(s, Alternate_s(s', s"))
Alternate_s(s,s') = Alternate_s(s',s)
Alternate_s(s,s) = s
end

Starting with names we can build a Pair of schemes, and we can put a collection symbol on the top of a scheme (Coll_s). Further we can build (s | s') for two given schemes s and s' with Alternate_s.

Examples of schemes:
sch1 = Coll_s(List, Pair_s(Inj(firstname), Inj(lastname))
= L(FIRSTNAME, LASTNAME)
sch2 = Pair_s(Inj(class), sch1) = (CLASS, L(FIRSTNAME, LASTNAME))
sch3 = Alternate_s(Inj(class), sch1) = (CLASS | L(FIRSTNAME, LASTNAME))

We represent a DTD by a function *type_n*, which gives for each name a corresponding scheme. There are user dependent names, which are described in general by the user and some system names, which are equal for all applications. We give only some example equations.

def
opers **type_n** (n: Name iff in(n, {ZAHL,TEXT,..}= false) ⟶ Scheme
axioms
type_n(result) = Pair_s(subject, mark)
type_n(pupil) = (firstname, lastname, Coll_s(List, result))
type_n(class) = List(pupil)
...
end

The following specification contains some useful simple operations.

def
opers **comp-no** (Scheme) ⟶ Nat // the number of components of a scheme
equal-s (Scheme, Scheme) ⟶ Bool // unspecified; simple equality relation
comp? (s: Scheme, s': Scheme) ⟶ Bool // each component of s occurs in s'
coll? (s: Scheme) ⟶ Bool // s is a scheme for a collection
red (s:Scheme iff coll?(s) = true) ⟶ Scheme
(a collection scheme is reduced by the topmost collection symbol)
coll-type (s:Scheme iff coll?(s)) ⟶ Coll-sym // the collection type of a collection
axioms cs: Coll-sym; s, s', s": Scheme; n: Name; t, t': Tabment
comp-no(Empty_s) = zero
comp-no(Coll_s(cs, s)) = comp-no(Inj(n)) = comp-no(Alternate_s(s, s')) = one
comp-no(Pair_s(s, s')) = comp-no(s) + comp-no(s')

if comp-no(s) = one & comp-no(s') = one then comp?(s, s') = equal-s(s, s')
if comp-no(s) = one then comp?(s, Pair_s(s', s")) = (comp?(s, s') or comp?(s, s"))
comp?(s, Empty_s) = equal-s(s, Empty_s)
comp?(Empty_s, s) = true
comp?(Pair_s(s, s'), s") = (comp?(s, s") and comp?(s', s"))

coll?(Coll_s(cs, s)) = true
coll?(Inj(n)) = false

coll?(Empty_s) = coll?(Alternate_s(s, s’)) = false
if comp-no(s)>one then coll?(s) = false
red(Coll_s(cs, s)) = s
coll-type(Coll(cs, s)) = cs
end

III. SPECIFICATION OF XML-DOCUMENTS

The following tabment specification is a generalization of the following concepts: number, text,…, set (relation), list (sequence),bag (multi-set), array, element (of a collection), optional element, (XML)-document and table.

def
sorts **Tabment**
opers **Empty_t** ⟶ Tabment
El_tab (Value) ⟶ Tabment // an elementary table (contains one value)
Empty (s: Scheme iff coll?(s)) ⟶ Tabment
Add (t1: Tabment, t2: Tabment iff red(type_t(t1)) = type_t(t2) or
coll_type(type_t(t1))=Any) ⟶ Tabment
Pair_t (Tabment, Tabment) ⟶ Tabment
Alternate_t (t: Tabment, s: Scheme) ⟶ Tabment
Tag0 (n: Name, t: Tabment iff type_n(n) = type_t(t)) ⟶ Tabment
type_t (Tabment) ⟶ Scheme
axioms n: Name; s, s’, s“: Scheme; t, t’, t1, t2, t3: Tabment; l: Letter, d: Digit, se: Separator, b: Bool
type_t(Empty_t) = Empty_s
type_t(El_tab(String_v(s)) = Inj(TEXT), …
type_t(El_tab(Bool_v(b))) = Inj(BOOL), type_t(El_tab(Bar))= Inj (BAR)
if coll?(s) then type_t(Empty(s)) = s
if t = Add(t1, t2) then type_t(t) = type_t(t1)
type_t(Pair_t(t1, t2)) = Pair_s(type_t(t1), type_t(t2))
type_t(Alternate(t, s)) = Alternate_s(type_t(t), s)
if t = Tag0(n, t’) then type_t(t) = Inj(n)
Pair_t(Empty_t, t) = Pair_t(t, Empty_t) = t
Pair_t(t1, Pair_t(t2, t3)) = Pair_t(Pair_t(t1, t2), t3)
Alternate_t(Alternate_t(t, s’), s“) = Alternate_t(t, Alternate_s(s’, s“))
if coll-type(type_t(t1)) = Set & red(type_t(t1)) = type_t(t2) = type_t(t3)
then Add(Add(t1, t2), t3) = Add(Add(t1, t3), t2)
if coll-type(type_t(t1)) = Bag & red(type_t(t1)) = type_t(t2) = type_t(t3)
then Add(Add(t1, t2), t3) = Add(Add(t1, t3), t2)
if type_t(t1) = Coll_s(Set, type_t(t2))
then Add(Add(t1, t2), t2) = Add(t1, t2)
if coll-type(type_t(t1)) = S1 & type_t(t2) = type_t(t3) = red(type_t(t1))
then Add(Add(t1, t2), t3) = Add(t1, t2)
end

Now, we illustrate the generating operations by examples:
Empty_t = <></>
El_tab(a) = <TEXT>a</TEXT>=<<TEXT:: a>>
El_tab(3) = <ZAHL>3</ZAHL>=<<ZAHL:: 3 >>,…

Tag0(n, <s> t </s>) = <n> <s> v </s> </n>

t1 = <s_{11},s_{12},...,s_{1n}> <s_{11}> v_{11} </s_{11}>
<s_{12}> v_{12} </s_{12}>
...
<s_{1n}> v_{1n} </s_{1n}>
</s_{11}, s_{12}, ...,s_{1n}>
t2 = <s_{21}, s_{22}, ..., s_{2m}> <s_{21}> v_{21} </s_{21}>
<s_{22}> v_{22} </s_{22}>
...
<s_{2m}>v_{2m}</s_{2m}>
</s_{21}, s_{22},...,s_{2m}>,
with comp-no(s_{ij}) = 1 for each i and j
Pair_t(t1, t2) = <s_{11}, s_{12},..., s_{1n}, s_{21},s_{22},...,s_{2m}>
<s_{11}>v_{11} </s_{11}>
<s_{12}> v_{12} </s_{12}>
...
<s_{1n}> v_{1n} </s_{1n}>
<s_{21}> v_{21} </s_{21}>
<s_{22}> v_{22} </s_{22}>
...
<s_{2m}>v_{2m}</s_{2m}>
</s_{11}, s_{12}, ..., s_{1n}, s_{21}, s_{22}, ..., s_{2m}>

Empty(Coll_s(C, s)) = <C(s)></C(s)>

t1 = <C(s)> <s> v_1 </s>
<s> v_2 </s>
...
<s> v_n </s>
</C(s)>
t2 = <s> v </s>
Add(t1, t2) = <C(s)> <s> v_1 :s>>
<s> v_2 </s>
...
<s> v_n </s>
<s> v </s>
</C(s)>

t = <s> v </s>
Alternate_t(t, s') = <s | s'> <s> v </s> </s | s'>

IV. DIFFERENCES BETWEEN XML AND SPECIFICATION

In the following, we shall name the objects of specification table and the XML-documents short documents.

1. To represent XML-documents we need not only names but also schemes as tags.
2. The specification does not distinguish between attributes and elements; an attribute is a special element. From abstract point of view there is no difference between attributes

and elements. If special elements are desired, they could be signed by a precceding "@", for example.

3. In the specification a tuple of several elements is distinguished from a sequence of these elements. On components of tuples we can access for example with names and numbers and on elements of collections with numbers.
4. A List of simple values like integers does not exist for example in the specification, but a list of integers "tagged" by INT can be considered as a table.
5. A tabment, which is a n-tuple, has exactly *n* children (components). An "XML-tuple" may have less (empty collection or ?) or more (for example: an X-document with type_n(X) = (A, B*) may have one A-child + five B-children) than *n* children.
6. A tabment, which is a collection of *n* elements (element in the set-theoretic sense), has exactly *n* children. A document *X* of *n* elements with type_n(X) = (A, B)* has for example *2n* children.
7. The specification knows additional basic collection types (Set, Bag, and Any).
8. Contrary to XQuery in the specification we distinguish consequently between a singleton and the element, which the singleton contains.

V. SPECIFICATION OF FORGET

The introduction of an operation *forget* enriches our XML-algebra. By *forget(t, ns)* all *n*-subtables of *t*, for each *n* of *ns* is omitted. The structuring of *t* remains unchanged. Because this removal goes recursively into arbitrary depth *forget* can be applied in some cases, where *stroke* is not strong enough. For example:
type_n(PERSONS) = M(PERSON), with
type_n(PERSON) = (NAME, LOC, M(HOBBY), MGR?, M(CHILD)),
type_n(NAME) = type_n(LOC) = type_n(HOBBY) = TEXT,
type_n(MGR) = type_n(CHILD) = PERSON
We will specify *forget* in such that for example the following holds:
type_n(forget(PERSONS, {LOC, HOBBY})) = M(PERSON) with
type_n(PERSON) = (NAME, MGR?, M(CHILD)),
type_n(NAME) = TEXT,
type_n(MGR) = type_n(CHILD) = PERSON

Especially, it is visible that by this removal of HOBBY the whole collection M(HOBBY) disappears. In the same way in the following specification by the removal of alternatives the whole alternative is removed. For example, if we forget *B* in *(A | B)* then not *(A | Empty_s)* but *A* results. In our opinion these design decisions simplify the usability of our XML-algebra, although they complicate the specification of our operations.
It holds for example:

<table>
<tr><td rowspan="3">forget(</td><td>M</td><td rowspan="3">, {A})</td><td rowspan="3">=</td><td>M</td></tr>
<tr><td>A | B</td><td>B</td></tr>
<tr><td>a
b</td><td>b</td></tr>
</table>

The above term *forget(persons, {HOBBY, LOC)})* can be expressed in XQuery by introduction of a recursive function similar to example 1.2.4.1 Q1 from [CFFRM02] in the following way:

```
define function forget2( element $e )
        returns element*
```

```
        {
         let $n := local-name( $e )
         return
           if ($n = "person")
           then
               <person>
                 { $e/name }
                 <mgr>{ forget2($e/mgr/person) }</mgr>
                  { for $c in $e/child
                  return {<child>{ forget2($c/person) }</child>}}
               </person>
           else ()
        }

<persons2>
          {
           forget2( document("persons.xml")/person)
          }
</persons2>
```

To specify *forget* we need a sort for names and an element relation for names.

sorts **Names**
opers **Empty-n** ⟶ Names // the Empty set of names
{ Name **}** ⟶ Names // a singleton of names
union-n (Names, Names) ⟶ Names // set theoretic union
axioms n: Name; ns, ns1, ns2, ns3 : Names
union-n(ns, Empty-n) = ns
union-n(ns1, union-n(ns2, {n})) = union-n(union-n(ns1, ns2), {n})
union-n(union-n(ns, {n}), {n}) = union-n(ns, {n})
union-n(ns1, ns2) = union-n(ns2, ns1)
union-n(union-n(ns1, ns2), ns3) = union-n(ns1, union-n(ns2, ns3))
end
opers **forget** (t: Tabment, ns: Names) ⟶ Tabment
(forget all *n*-subtables from *t*, for each *n* from *ns*)
forget_s (s: Scheme, ns: Names) ⟶ Scheme
(forget all names from *ns* in *s*)
in-n (Name, Names) ⟶ Bool
axioms n, n’: Name; ns: Names; cs: Coll-sym; s, s’: Scheme; t, t’: Tabment
in-n(n, Empty-n) = false
in-n(n, union(ns, {n’}) = (in-n(n, ns) or equal-n(n, n’))

forget_s(Empty_s, ns) = Empty_s
if in-n(n, ns) then forget_s(Inj(n), ns) = Empty_s
if in-n(n, ns) = false then forget_s(Inj(n), ns) = Inj(n)
if forget_s(s, ns) != Empty_s
then forget_s(Coll_s(cs, s), ns) = Coll_s(cs, forget_s(s, ns))
if forget_s(s, ns) = Empty_s then forget_s(Coll_s(cs, s), ns) = Empty_s
forget_s(Pair_s(s, s‘), ns) = Pair_s(forget_s(s, ns), forget_s(s‘, ns))
if forget_s(s, ns) = Empty_s then forget_s(Alternate_s(s, s’), ns) = forget_s(s’, ns)
if forget_s(s, ns) != Empty_s & forget_s(s’, ns) != Empty_s
then forget_s(Alternate_s(s, s’), ns) =

= Alternate_s(forget_s(s, ns), forget_s(s’, ns))

forget(Empty_t, ns) = Empty_t
if type_t(t) = Inj(n) & in-n(n, ns) then forget(t, ns) = Empty_t
if type_t(t) = Inj(n) & in-n(n, ns) = false & t = El_tab(v) then forget(t, ns) = t
if coll?(s) & forget_s(s, ns) != Empty_s
then forget(Empty(s), ns) = Empty(forget_s(s, ns))
if type_t(t) = s & forget_s(s, ns) = Empty_s then forget(t, ns) = Empty_t
if t = Add(t‘, t“) & forget(t“, ns) != Empty_t
then forget(t, ns) = Add(forget(t‘, ns), forget(t“, ns))
if t = Add(t‘, t“) & forget(t“, ns) = Empty_t
then forget(t, ns) = forget(t‘, ns)
forget(Pair_t(t, t‘), ns) = Pair_t(forget(t, ns), forget(t‘, ns))
if forget_s(type_t(t), ns) != Empty_s & forget_s(s, ns) != Empty_s
then forget(Alternate_t(t, s), ns) = Alternate_t(forget(t, ns), forget_s(s, ns)) &
if forget_s(type_t(t), ns) = Empty_s
then forget(Alternate_t(t, s), n)) = Empty_t
if forget_s(s, ns) = Empty_s &
then forget(Alternate_t (t, s), ns) = forget(t, ns)
if t = Tag0(n, t‘) & forget(t’, ns) != Empty_t & in-n(n, ns) = false
then forget(t, ns) = Tag0(n, forget(t‘, ns))
if t = Tag0(n, t‘) & forget(t’, ns) = Empty_t then forget(t, ns) = Empty_t
if t = Tag0(n, t‘) & in-n(n, ns) then forget(t, ns) = Empty_t

end