

---

# A Type-safe Macro System for XML

---

Thomas Perst  
University Trier

Helmut Seidl  
University Trier

---

## **Abstract**

XML is originally designed for the use in document processing. Every classical document processing system supports a mechanism which is called *macro*. In general, a macro is a named collection of actions. These actions are executed whenever the name of the macro appears in a document instance. We describe a very small but powerful macro language for XML. The presented language supports macros with arguments. As a main result we will show that for a macro expansion  $H$  and a recognizable set  $R$  the set  $H^{-1}(R)$  of trees  $t$ , whose macro expansion  $H(t)$  are in  $R$ , is recognizable. This implies that the typechecking problem for macro expansion is decidable. A major drawback of this approach is the time complexity. Therefore we designed a practical typechecking discipline. We equip every macro definition with a type expression describing the input forests. On the basis of this type information the macro processor computes an output type for the macro. Having inferred this output type, we aim at extending the given output DTD to a DTD for admissible input possibly containing unexpanded macros.

## A Type-safe Macro System for XML

### **Table of Contents**

1 Introduction.....	1
2 The Macro Language.....	3
3 Trees and Forests.....	6
4 Expressiveness.....	6
5 Practical Typechecking.....	11
6 Conclusion and Future Work.....	14
Bibliography.....	14
The Authors.....	15





# A Type-safe Macro System for XML

Thomas Perst and Helmut Seidl

## § 1 Introduction

XML is a language for storing structured data in a sequential format. It is originally designed for the use in document processing[11] and it is believed to become the de facto data exchange format for the Internet[9].

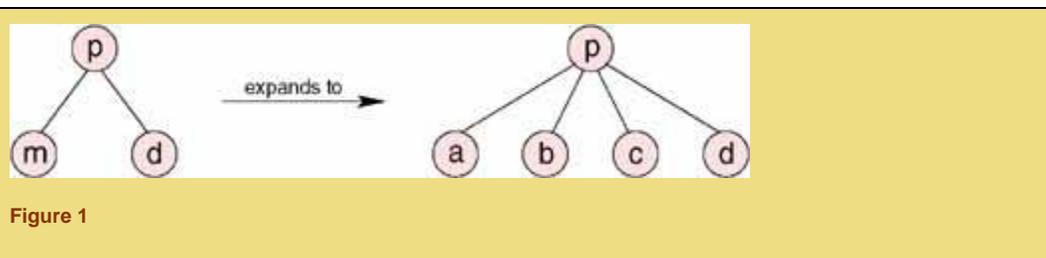
All classical document processing systems support a mechanism which is called *macro*. A macro is a named collection of actions. It is often nothing but an abbreviation for a longer piece of text which will frequently appear in a document. In this case macro processing becomes text replacement: whenever the macro name appears in the document it will be automatically replaced by the defined piece of text. Macros get more powerful by having additional parameters. Then macros are expanded dependent on the actual input.

In the case of XML, documents are structured by inserting *markup* into the text, indicating the start and the end of a logical component. Each logical component, called *element* is enclosed between a *start-tag* and an *end-tag*. We can consider an XML document as a tree whose nodes are labeled with tags. Besides, the concept of *entities* is known in XML. These are a mechanism for modularization of a document[11][6]. Entities do not have arguments. Their references in a document are replaced by constant text. As stated in[5], XML entities are only an abbreviation device. Our aim is to introduce macros with arguments.

A macro for XML is a named description of a forest. Whenever a parser makes a document available to an application the macro is expanded to the defined forest. Assume we define macro *m* as the forest

```
<a/><b/><c/>
```

then a document instance *p* and its expansion might look like:



*m* is replaced by the forest `<a/><b/><c/>`. We can think of parameters as element content of the macro. Each parameter itself is a forest. In the macro definition we refer to a parameter with a special element `<par p="i" />` where *i* is the number of the input forest we want to refer to. If we define macro *n* as

```
<b/><par p="2" /><h><par p="1" /></h>
```

then we get:

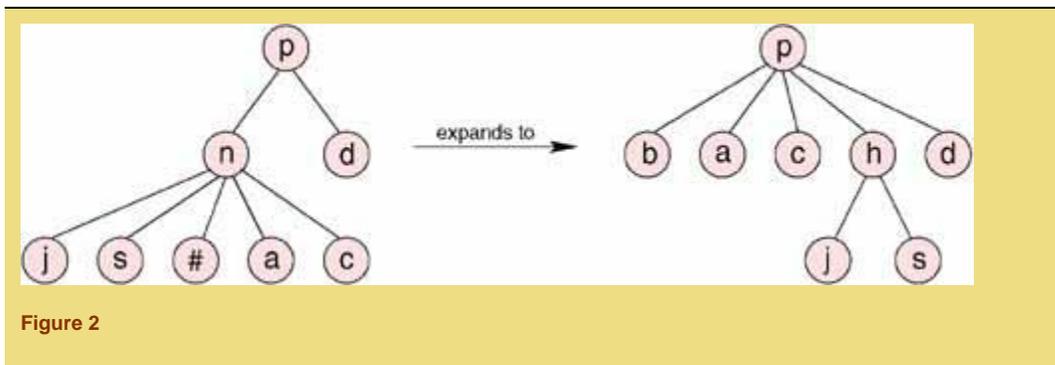


Figure 2

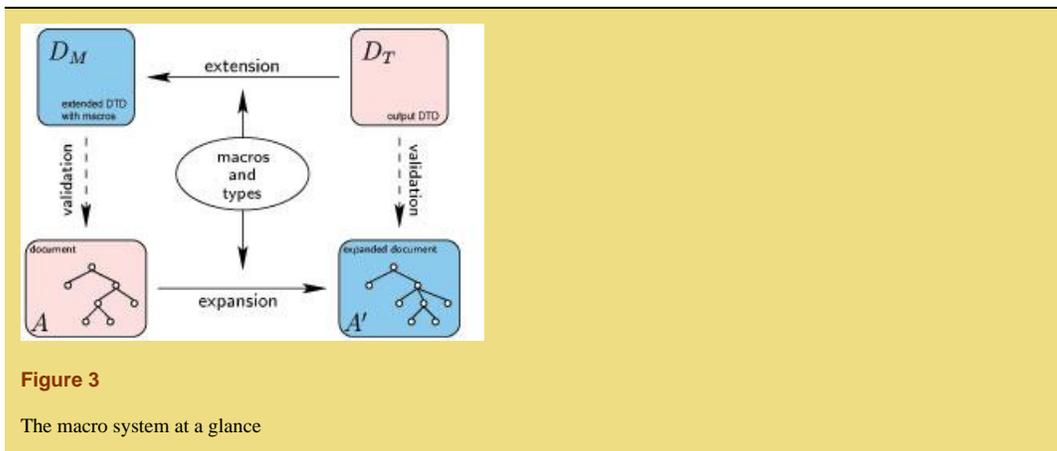
While XML is a markup language we therefore can conceive macros as higher-order markup[3] in the following sense: given a possibly empty input forest, a macro is evaluated to an output forest, consisting of text and normal markup.

As XML is a data exchange format, some sort of agreement is necessary so that one application is able to understand the data produced by another. This agreement is known as *document type definition*, DTD for short. A DTD is a context-free grammar describing a set of trees.

In general, it is easy to check if a document is of a given type. Normally this is done at parsing time and called validation. Often documents are automatically generated by programs. Then we want to check if every document produced by the program is of a given type. This process is called *typechecking problem*. For a program  $P$  and an output type  $\tau$  it asks whether all data instances produced by  $P$  are of type  $\tau$ . An important application class are *transformers*. They map XML trees to XML trees. Milo, Suci and Vianu focus on the typechecking problem in[10] from a different point of view. Given a description of valid outputs and a program that transforms documents, determine the set of inputs which are guaranteed to produce valid outputs only. Macro expansion constitutes a simple form of tree transformation. We will show that for every macro expansion and a given output type this set of inputs is computable. This result implies that the typechecking problem for macro expansion is decidable. A drawback of this approach is the exponential time it needs.

Therefore we have chosen another approach for an implementation. We construct in polynomial time a DTD for a large set of inputs which is guaranteed to produce valid outputs only, given that we implement the macro system as a typed programming language. According to this approach we equip a macro definition with a type for each input forest. Then it is possible to infer the output type of the macro. Having an output type for each macro we extend the given output DTD to a DTD for admissible input possibly containing unexpanded macros. In this way the XML parser is able to validate the document before expanding the macros.

Our implementation[14] uses *fxp*[6] as our XML parser and is written in the functional programming language *SML*[7][8].



In the following section we present a very small but powerful macro language for XML. Having introduced some definitions in the forth section, the expressiveness and the relation to *k*-pebble transducers is shown. Then we present our practical typechecking algorithm.

## § 2 The Macro Language

In this section we develop some examples of programming in our small macro language for XML. The language uses XML for both, defining and calling macros.

Suppose that we are using an XML based mailtool to create and read our mail. The following document type definition (DTD) specifies how the elements inside a mail should relate to each other.

```
<!ELEMENT email      (header,body,signature) >
<!ELEMENT header    (from,to+,cc*,subject,reply-to?) >
<!ELEMENT from      (#PCDATA) >
<!ELEMENT to        (#PCDATA) >
<!ELEMENT cc        (#PCDATA) >
<!ELEMENT subject   (#PCDATA)*>
<!ELEMENT reply-to  (#PCDATA) >
<!ELEMENT body      (#PCDATA) >
<!ELEMENT signature (#PCDATA) >
```

A mail consists of a header, a body and a signature. The header contains information about who authored the message, the intended recipients and the subject of the message. The body contains the information the sender is trying to communicate. And signature is a personal footer which is added to each outgoing message[13].

Now let us define two very simple macros:

```
<def macro='bart'>bart@springfield-elementary.edu</def>
<def macro='f'><from><bart/></from></def>
```

Every time these macros are used in a document instance they will produce the same output because they do not depend on any input. The new macro element *f* does not have any content, it is an empty element. Each macro call `<f/>` will be replaced by `<from>bart@springfield-elementary.edu</from>`.

In a macro definition every already defined macro can be used. The macro definition of *f* uses macro *bart* to produce the content of *from*.

The next macro, *family*, expands to a forest consisting of three *to* elements. Creating a new message, this macro can be used as a mail alias to send the message to every family member.

```
<def macro='family'>
  <to>homer.simpson@snpp.com</to>
  <to>marge@freemail.org</to>
  <to>lisa@freemail.org</to>
</def>
```

As stated before, if macros do not have arguments macro processing becomes text replacement. Now we want to look at macros with arguments. Normally, the *from* and *reply-to* fields are the same in every mail we send. Therefore it would be useful to define a macro that takes only the recipients and the subject as input while the other fields of the header are produced automatically. Such a macro looks like

```
<def macro='h'>
  <header>
    <f/>
    <par/>
    <subject><par p='2' /></subject>
    <reply-to><me/></reply-to>
  </header>
</def>
```

Macro *h* takes two forests as arguments. These forests will be inserted at the corresponding places. The first argument contains the recipients and the second argument the subject of the message. The special element `<par p="i" />` is a reference to the forest with number *i*. The input forests are numbered from left to right starting with *1*. The element `<par />` is an abbreviation for `<par p="1" />`.

A call of this macro can be

```
<h><family/><sep/>Radioactive Man</h>
```

The macro element *family* expands to several *to* elements. The empty element *sep* is a reserved element of the macro language. It is used to separate the different input forests. The second argument is the text "Radioactive Man". During expansion this text is placed between the start- and end-tag of the subject element.

For our practical typechecking algorithm it is necessary to type the input forests of the macro. The type is given in the syntax of XML content models. On the basis of this type information the macro processor computes a type expression for the output forest.

According to the DTD it is easy to see that after the *from* element one or more *to* elements follow, then optionally *cc* elements followed by a subject and optionally a reply-to address. Therefore the type of the input for macro *h* is  $(to+, sep, \#PCDATA)$ . The typed macro definition is:

```
(to+, sep, #PCDATA) <def macro='h'>
  <header>
    <f/>
    <par/>
    <subject><par p='2' /></subject>
    <reply-to><me/></reply-to>
  </header>
</def>
```

An additional element of the macro language handles attributes. The value of an attribute can be set by an attribute in the macro call. The macro *test* creates an element *style* with the attribute *color*. The value of *color* is the value of attribute *col* of the macro call.

```
EMPTY <def macro='test'>
  <style color='$col'>Lisa Simpson</style>
</def>
```

For example, the macro call `<test col='yellow' />` will produce `<style color='yellow'>Lisa Simpson</style>`.

A possible document instance with the unexpanded macros *family* and *h* might look like:

```
<?xml version="1.0" encoding="L1"?>
<!DOCTYPE email SYSTEM "Email.dtd">
<!-- open file MailMacros.xml -->
<!-- and read the macro definitions -->
<?UsePackage MailMacros.xml?>
<!-- begin mail -->
<email>
  <h><family/><sep/>Radioactive Man</h>
  <body>
    None of the new superheroes can hold a candle
    to "Radioactive Man".
  </body>
  <signature>
    Ay Caramba!
  </signature>
</email>
<!-- end mail -->
```

The call of macro *h* uses `<sep/>` to separate the two input forests. The macro processor expands the document to:

```
<email>
  <header>
    <from>bart@springfield-elementary.edu</from>
    <to>homer.simpson@snpp.com</to>
    <to>marge@freemail.org</to>
    <to>lisa@freemail.org</to>
    <subject>Radioactive Man</subject>
    <reply-to>bart@springfield-elementary.edu</reply-to>
  </header>
  <body>
    None of the new superheroes can hold a candle
    to "Radioactive Man".
  </body>
  <signature>
    Ay Caramba!
  </signature>
</email>
```

In addition the processor extends the original DTD. In this generated DTD the macros are used as normal elements. Macro *h* for example produces a tree of type *header*. It follows that *h* can be used in every context where *header* can be used.

```
<!ELEMENT email      ((header|h),body,signature) >
<!ELEMENT header    ((from|f),(to+|family),cc*,subject,reply-to?)
>
<!ELEMENT from      (#PCDATA|bart)* >
```

```

<!ELEMENT to      (#PCDATA | bart)* >
<!ELEMENT cc      (#PCDATA | bart)* >
<!ELEMENT subject (#PCDATA | bart)* >
<!ELEMENT reply-to (#PCDATA | bart)* >
<!ELEMENT body    (#PCDATA | bart)* >
<!ELEMENT signature (#PCDATA | bart)* >
<!ELEMENT sep     EMPTY >
<!ELEMENT bart    EMPTY >
<!ELEMENT f       EMPTY >
<!ELEMENT family  EMPTY >
<!ELEMENT h       (#PCDATA | sep | to | bart)* >

```

With this DTD it is possible to validate a document with unexpanded macro calls. A document conforming to the extended DTD then is guaranteed to be expanded into a document which conforms to the original DTD.

### § 3 Trees and Forests

A semistructured data instance is a tree whose nodes are labeled with tags. We use a simple syntax to represent these trees.

Let  $\Sigma$  be an alphabet whose symbols are the tags of the trees. The sets  $T_\Sigma$  of trees  $t$  and  $F_\Sigma$  of forests  $f$  over  $\Sigma$  are given as follows:

$$\begin{aligned}
 t &::= a\langle f \rangle, & a \in \Sigma \\
 &| \text{"s"} \\
 f &::= t_1 \dots t_n, & n \geq 0
 \end{aligned}$$

Figure 4

For a tree  $t = a\langle f \rangle$  this occurrence of  $a$  is called the *root node* of  $t$  and  $\text{sym}(t)=a$  is the *symbol* of that node or the tag with which  $t$  is labeled.  $\text{succ}(t)=f$  are the children or successors of  $t$ .

For  $t = \text{"s"}$  the tree only consists of the *character data*  $s$  and does not have any children. Trees of this form are also called *text nodes*.

The sets  $T_\Sigma$  and  $F_\Sigma$  are given by the *document type definition* (DTD). The DTD is a context-free language which describes for each tree  $t$  the type of the content. These content specifications are regular expressions defining  $\text{succ}(t)$ .

### § 4 Expressiveness

Milo, Suciú and Vianu study in [10] the typechecking problem for XML transformers: given a description of valid outputs -- through some recognizable tree set -- and a program that transforms documents, determine the set of inputs which are guaranteed to produce valid outputs only.

In order to model the XML transformers, they introduce the concept of  $k$ -pebble transducers. These transducers can express many transformations over unranked trees over a given finite alphabet. For example a 1-pebble transducer  $T_C$  can copy the input tree to the output, or a transducer  $T_E$  maps a tree to an exponentially larger tree. The authors point out that every top-down tree transducer [12] can be expressed as a 1-pebble transducer [10]. Macro expansion constitutes another simple form of tree transformation. Surprisingly, this expansion cannot be implemented by using  $k$ -pebble transducers.

**Theorem** *The macro expansion cannot be expressed with  $k$ -pebble transducers.*

**Proof** Consider the macro  $m$  whose expansion is defined as the *duplication* of the input forest:

$$m : \mathcal{F}_\Sigma \longrightarrow \mathcal{F}_\Sigma, m(f) = ff.$$

Figure 5

It follows that the  $n$ -fold composition

$$m(m(\dots(m(f))\dots))$$

Figure 6

produces an output forest  $f^n = f \cdot f$ , which has length exponential in the length of the input forest  $f$ . Using the function *encode* from [10] that encodes unranked trees and forests into binary trees, the resulting forest  $f^n$  is mapped to a tree of exponential depth. On the other hand, by proposition 3.8 in [10], the set of output trees produced by a  $k$ -pebble transducer for a tree  $t$ , always contains some tree  $s$  whose depth is polynomial in the size of  $t$ . We conclude that the output  $f^n$  cannot be produced by any  $k$ -pebble transducer.

Accordingly, the following results do not follow from the analogous results for  $k$ -pebble transducers in [10].

**Theorem** Assume we are given a recognizable set  $R$ .

1. Then the set  $H_{\cdot 1}(R)$  of trees  $t$ , whose macro expansion  $H(t)$  are in  $R$ , is recognizable.
2. Given a deterministic bottom-up automaton for  $R$ , a deterministic bottom-up automaton for  $H_{\cdot 1}(R)$  can be constructed in exponential time.

**Proof** The key idea is to enhance the bottom-up automaton for  $R$  to a *monoid automaton*. Let  $A_R$  be any deterministic bottom-up automaton for  $R$

$$A_R = (P_R, Q_R, q_0, F_R, Up_R, Side_R)$$

Figure 7

The sets  $P_R$  and  $Q_R$  represent the tree and forest states, respectively.  $q_0$  is the initial state and  $F_R$  are the final states. The side-transition  $Side_R$  describes how a state is assigned to a forest. The up-transition  $Up_R$  describes how a tree state is assigned to a tree. For a more precise definition see [6].

Let  $M = Q_R \rightarrow Q_R$  be the monoid of functions from  $Q_R$  into itself, equipped with the operation

$$\cdot : M^2 \longrightarrow M, f_1 \cdot f_2 = f_2 \circ f_1 \text{ (function composition).}$$

Figure 8

Let  $\mu$  denote the homomorphism from  $i P_{R^*}$  into the monoid  $M$  which maps tree states of the given automaton  $A_R$  to  $M$  such that

$$\mu(p)(q) = Side_R(q, p).$$

Figure 9

This reflects the natural relation between tree and forest states. In order to assign a state to a forest, a tree state is first assigned to each tree of the forest. Traversing this word of tree states from left to right and performing a side transition at each step, a forest state is obtained from an initial state[6]. The homomorphism  $\mu$  describes the side transition which can be performed at a tree state  $p \in P_R$ , even if we do not have any information about the already performed transitions. We construct automaton  $A_I$  where we use the monoid  $M$  for forest states and side transitions. Thus,

$$A_1 = (P_R, M, m_0, F_1, Up_1, Side_1).$$

Figure 10

The new *side transition* of  $A_I$  is the composition of the reached forest state  $m \in M$  and  $\mu(p)$ . An *up transition* for a forest state  $m$  and symbol  $a$  yields a tree state for the tree with label  $a$ . Thus, the up transition of  $A_I$  first computes the actual forest state  $q' \in Q_R$  by applying the monoid element (which is a function  $P_R \rightarrow P_R$ ) to the initial state  $q_0$ . From  $a$  and the resulting state  $q'$  the new tree state is obtained by the up transition of  $A_R$ :

$$\begin{aligned} Side_1(m, p) &= m \cdot \mu(p) \\ Up_1(m, a) &= Up_R(m(q_0), a) \end{aligned}$$

Figure 11

The computation starts with forest state  $m_0 = \mu(\epsilon)$  which is indeed the identity function 1 in monoid  $M$ . Automaton  $A_I$  accepts if a forest state  $m$  is reached which describes a series of transitions ending in an accepting state of  $A_R$ .

$$\begin{aligned} m_0 &= \mu(\epsilon) = 1, \quad 1 \in M \\ F_1 &= \{m \mid m(q_0) \in F_R\} \end{aligned}$$

Figure 12

This automaton exactly accepts  $R$  because it simulates the computation of  $A_R$  through the transition monoid  $M$ . We now construct from  $A_I$  the automaton  $A_2$  for the *pre-image*  $H^{-1}(R)$ :

$$A_2 = (M \cup \{m_\#, \diamond\}, Q_2, m_0, F_1, Up_2, Side_2)$$

Figure 13

This automaton has to accept the set of trees  $t$ , whose macro expansion  $H(t) \in R$ . Assume  $t$  is given as

$$\alpha \langle f_1 \# f_2 \# \dots \# f_k \rangle$$

Figure 14

with macro  $\alpha$  and  $f_1, \dots, f_k$  forests. Assume for example, that the macro  $\alpha$  is defined as follows:

$$e_\alpha \equiv x_k \langle c_1 \rangle \dots c_k \langle x_1 \rangle$$

Figure 15

where  $x_i \langle \rangle$ ,  $1 \leq i \leq k$ , are formal parameters which are to be replaced by the actual parameters  $f_i$  when  $\alpha$  is expanded. For simplicity, we assume that we are given  $n$  empty trees  $x_i \langle \rangle$  signaling the formal parameters. The automaton  $A_2$  has to simulate this expansion and accept only those trees whose expansion is in  $R$ . Tree states are elements in  $M$ . The new state  $m_\#$  signals that the computation has reached the delimiter symbol  $\#$  separating parameter forests. State  $\diamond$  is an error state. Forest states are tuples of elements in  $M$ . Let  $r$  be the maximum arity of all used macros then

$$Q_2 \subseteq M^{\leq r} \cup \{\diamond\}.$$

Figure 16

The forest states are chosen as tuples because we need to store the monoid elements corresponding to each actual parameter forest separately. The computation starts with  $m_0 \in M$ , the same initial state as in  $A_1$ . The automaton accepts and stops in the same states as  $A_1$ . Note that  $A_2$  still has to accept  $R$ . This is the case if no macros at all appear in the document. The *side transition* collects the input forests in the following way: If a "normal" tree state  $p \in M$  is reached while traversing a forest from left to right, the right-most element of the tuple is replaced by a new state. This new state is computed by a side transition of  $A_1$ . On the other hand, if the delimiter symbol is reached, the tuple itself is extended by a new element. This element is an initial forest state of  $A_1$ . Suppose there are more than  $r$  forests, the new state will be the error state  $\diamond$ . If the delimiter  $\#$  does not appear in the forest,  $Side_2$  shows exactly the same behaviour as the side transition of automaton  $A_1$ .

$$\begin{aligned} Side_2(s, m_\#) &= \begin{cases} \mu(\epsilon) :: s, & |s| < r \\ \diamond, & \text{else} \end{cases} \\ Side_2(m :: s, p) &= (m \cdot \mu(p)) :: s = (Side_1(m, p)) :: s \\ Side_2(\diamond, y) &= \diamond, \text{ for all } y \end{aligned}$$

Figure 17

An *up transition* for a forest state and a symbol yields a new tree state. For a "normal" symbol  $a$  and a forest state  $m$ ,  $Up_2$  simulates the behaviour of automaton  $A_1$ . If  $A_2$  is once run into the error state  $\diamond$ , the automaton will be trapped in this state. The delimiter symbol  $\#$  is handled like an empty element and leads to state  $m_\#$ . The most interesting case of  $Up_2$  is the macro expansion. Then the expansion  $e_\alpha$  of macro  $\alpha$  has to be evaluated. This is done by  $[\cdot] \rho: Q_2 \rightarrow M$ .

$$\begin{aligned}
 Up_2(m_1 \dots m_k, \alpha) &= [e_\alpha] \{x_1 \mapsto m_1, \dots, x_k \mapsto m_k\} \\
 Up_2(m, a) &= \mu(Up_1(m, a)) \\
 Up_2(\diamond, x) &= \diamond, \text{ for all symbols } x \\
 Up_2(m_0, \#) &= m_\#
 \end{aligned}$$

Figure 18

$[\cdot] \rho$  simulates the expansion of a macro  $\alpha$ .  $\rho$  is an environment which defines the mapping from formal parameters to forest states. If the expansion simply consists of a formal parameter  $x_i$  the corresponding forest state is returned. If the expansion is a forest  $t_1 \dots t_n$  the trees are recursively evaluated by  $[\cdot] \rho$  where the results are then composed by means of the monoid operation. For a tree  $a \langle f \rangle$  the expansion  $[\cdot] \rho$  is applied to the forest  $f$  and then the up transition of automaton  $A_I$  is returned. Note that an expansion of a constant tree leads to the same tree state as in  $A_I$ .

$$\begin{aligned}
 [x_i] \rho &= \rho(x_i), \text{ for all } 1 \leq i \leq k \\
 [t_1 \dots t_n] \rho &= [t_1] \rho \cdot \dots \cdot [t_n] \rho \\
 [a \langle f \rangle] \rho &= \mu(Up_1([f] \rho, a))
 \end{aligned}$$

Figure 19

We verify by induction over the height of a tree  $t$  and forest  $f$  that every macro is correctly expanded, macro calls with too many arguments lead to the error state, and that for every expanded tree or forest only those states are computed which were computed in  $A_I$ . From the construction of  $M$  follows immediately that the bottom-up automaton for  $H_I(R)$  can be constructed in exponential time.

In general, typechecking is the problem of verifying that all data instances produced by a program are valid with respect to an output type. For a program

$$P : X \longrightarrow Y$$

Figure 20

- taking an input  $x \in X$  and returning the output  $P(x)$  - let  $\tau_1 \subseteq X$  be some input type and  $\tau_2 \subseteq Y$ ; then the typechecking problem asks whether

$$\forall x \in X, x \in \tau_1 : P(x) \in \tau_2.$$

Figure 21

The macro system presented in this paper maps XML trees to XML trees. Therefore the program has functionality:

$$H : \mathcal{T}_\Sigma \longrightarrow \mathcal{T}_\Sigma.$$

Figure 22

The mapping is given by the set of macro definitions. We conclude:

**Theorem** *The typechecking problem for macro expansion is decidable.*

**Proof** Let  $H$  be a macro expansion. Let  $R$  be a regular type; then the typechecking problem asks for an input  $t$  whether

$$H(t) \in R.$$

Figure 23

The previous theorem shows that  $H^{-1}(R)$  is accepted by a finite automaton  $A$ . Therefore the typechecking problem can be reduced to the question whether

$$t \in L(A)$$

Figure 24

where  $L(A)$  is the language accepted by  $A$ . This problem is decidable because membership in a regular language is decidable[2].

## § 5 Practical Typechecking

In the previous section we have proved that the typechecking problem for macro expansion is decidable. We also have shown that we can construct a bottom-up automaton for the pre-image of a macro expansion in exponential time. The time complexity is a major drawback for a practical macro system.

We now try to establish a typing discipline for macros. While Lämmel and Lohmann represent changes in the formats of documents as stepwise transformation on the underlying DTD[4], we have chosen a different approach. For a macro expansion  $H$  and a given DTD  $d$ , we try to exhibit a large set of inputs which is guaranteed to produce valid outputs only. Therefore we aim at extending the given DTD  $d$  to a DTD  $d'$ . This DTD  $d'$  is a description for admissible input possibly containing unexpanded macros.

The macro system presented in this paper maps XML trees to XML trees.

$$H : \mathcal{T}_\Sigma \longrightarrow \mathcal{T}_\Sigma.$$

Figure 25

The mapping is given by the set  $M$  of macro definitions. Each macro is defined as a pair  $(m, \sigma)$ .  $m$  describes the expansion and  $\sigma$  denotes the type expression of the input forest on which the expansion works.

In XML, a *type expression*  $\tau$  is either `EMPTY` or `ANY` or a regular expression  $r$  over the alphabet  $\Sigma \cup \{\#PCDATA\}$ . The set  $\Theta$  of type expressions  $\tau$  is given as follows:

$$\begin{aligned} \tau &::= \text{EMPTY} \mid \text{ANY} \mid r \\ r &::= a \in \Sigma \mid \#PCDATA \mid (r_1|r_2) \mid (r_1, r_2) \mid \\ &\quad (r)? \mid (r)^+ \mid (r)^* \end{aligned}$$

Figure 26

where  $r_1$  and  $r_2$  are regular expressions over  $\Sigma \cup \{\#PCDATA\}$ . Note that a type expression constrains the element's content. They constrain which element types can appear as children of the element[11].

So they describe only the top level or root nodes of successors and not the complete type of a tree or forest. Elements of a type expression  $\tau$  are the words of the language  $L(\tau)$  which is defined inductively on the structure of  $\tau$ :

$$\begin{aligned}
 L(\text{EMPTY}) &= \{\text{EMPTY}\} \\
 L(\text{ANY}) &= \mathcal{F}_\Sigma \\
 L(a) &= \{t \mid t \in \mathcal{T}_\Sigma \text{ and } a = \text{sym}(t)\} \\
 L(\#\text{PCDATA}) &= \mathcal{U}^* \\
 L((r_1|r_2)) &= L(r_1) \cup L(r_2) \\
 L((r_1, r_2)) &= \{f_1 f_2 \mid f_1 \in L(r_1), f_2 \in L(r_2)\} \\
 L((r)?) &= \{\epsilon\} \cup L(r) \\
 L((r)+) &= \bigcup_{i=1}^{\infty} L((r)\{i\}) \\
 L((r)*) &= \{\epsilon\} \cup L((r)+)
 \end{aligned}$$

Figure 27

In our case, the type expression  $\sigma$  constrains the input forest of a macro  $m$ . From this expression  $\sigma$  and the description of the expansion we infer a type expression  $R(m)=\sigma'$  for the output of a macro  $m$ . For that reason, we infer the type expression  $\text{typ}()$  for the forest describing the macro expansion. The empty forest is mapped to `EMPTY`, a text node to `#PCDATA`, a tree to the symbol of the root node, and a forest recursively to the sequence of tree type expressions.

$$\begin{aligned}
 \text{typ}(\epsilon) &= \text{EMPTY} \\
 \text{typ}("s") &= \#\text{PCDATA} \\
 \text{typ}(a\langle f \rangle) &= \text{sym}(t) = a \\
 \text{typ}(f) &= \text{typ}(t_1 \dots t_n) = (\text{typ}(t_1), \dots, \text{typ}(t_n))
 \end{aligned}$$

Figure 28

For a tree  $t=x_i\langle \rangle$ ,  $i \in \{1,2,3,\dots\}$  the type expression is defined as follows

$$\text{typ}(t) := \text{ptyp}(i),$$

Figure 29

where  $\text{ptyp}: \{1,2,3,\dots\} \rightarrow \Theta$  is a function which maps the formal parameter to its type:

$$ptyp(i) = \begin{cases} \text{EMPTY,} & \text{if } \sigma = \text{EMPTY} \\ \text{ANY,} & \text{if } \sigma = \text{ANY or } \mathcal{I}(m) = \epsilon \\ \tau_i, & \text{if } \sigma = (\tau_1, \text{sep}, \dots, \text{sep}, \tau_k), \\ & \forall 1 \leq j \leq k : \text{sep} \notin \tau_j, \tau_j \in \Theta \end{cases}$$

Figure 30

In the definition of macro  $m$  unexpanded macros  $(n_i, \sigma_i), \dots, (n_k, \sigma_k)$  can appear. We can assume that the type expressions  $R(n_i)$  of their output are already evaluated. If  $t = n_i \langle f \rangle$  is a macro call, i.e.  $sym(t) \in M$ , and if  $L(yp(f)) \subseteq L(\sigma_i)$  then the type of  $t$  is

$$typ(t) = \mathcal{R}(n_i).$$

Figure 31

Having inferred the type expression  $R(m) = \sigma'$  for macro  $m$ , we check if  $\sigma'$  occurs somewhere in the content type definitions of DTD  $d$ . As stated before, a DTD describes for each element, the element's content by a so-called *content model*. Our type expressions are an extension of the content model because we do not distinguish between element content and mixed content. But it is easy to see that there is a mapping from type expressions to content models. We check for every content model  $c$  whether

$$\sigma' \subseteq c.$$

Figure 32

For each sub expression of  $c$  the inclusion is tested. If this test provides true then the macro produces a forest of this type and we can compose the tested sub expression and the macro by or without changing the semantics of the content model. Content models for element content are given as deterministic finite automata (DFA). This implies that the test can be done in polynomial time because the complement of a DFA can be computed in polynomial time. Then the inclusion test can be reduced to

$$\sigma' \cap \bar{c} = \emptyset.$$

Figure 33

For example, let  $c$  be given as  $(c_1, (c_2/c_3))$ . If  $\sigma' \subseteq c_2$ , then we can extend  $c$  to  $(c_1, ((c_2/m)/c_3))$  where  $m$  is the name of macro  $m$ . In this way, we can build up an extended DTD  $d'$  in polynomial time for admissible input for the macro expansion  $H$ . It follows as a consequence that

$$\forall t \in d' : H(t) \in d$$

Figure 34

In addition, our macro system can deal with nested macros. This means, the input forest of a macro can be generated by another macro. The order of evaluation is bottom-up. Accordingly, the innermost macro is expanded at first. With the resulting forest the next one and so on until the outermost macro

is expanded. In order to reflect nested macros in the automatically generated DTD, the content type definitions of macros are also extended.

As stated in a previous section, every XML tree can have an arbitrary number of attributes. The macro processor uses *fxp*'s programming interface[6] to handle attributes. The used attributes and their values are validated at expansion time by *fxp*.

## § 6 Conclusion and Future Work

We have presented several examples how to define macros with our macro language. We consider this macro language suitable for document processing systems. The language is very small and authors or programmers need not learn a completely new syntax, as the macro language uses XML syntax.

We have also shown for the presented macro system that the typechecking problem is decidable, and that the pre-image of an expansion can be computed in exponential time.

For the practical use of the macro system we designed a typechecking discipline which computes for every macro an output type. With this type, it is possible to generate an extended DTD describing inputs that produce valid documents with respect to a given output DTD. Using a DTD-aware editor, it is also possible to determine which macros are available and which macros are allowed in the actual context. Because of the extended DTD macro elements can be handled like normal markup. Beyond the immediate focus on typechecking with DTDs, it remains for future work to extend the results also to other schema languages like XML Schema.

---

## Bibliography

- [1] James Clark and Steve DeRose, editors. *XML Path Language (XPath) 1.0*. W3C, November, 1999. Available online <http://www.w3.org/TR/xpath>.
- [10] Tova Milo, Dan Suciu and Victor Vianu. Typechecking for XML Transformers. *Journal of Computer and System Science*, 2002. To appear, available online <http://www.cs.washington.edu/homes/suciu/files/paper.pdf>.
- [11] W3C. *Extensible Markup Language (XML) 1.0*, second edition, 6 October 2000. Available online <http://www.w3.org/TR/2000/REC-xml-20001006>.
- [12] Zoltán F löp and Heiko Vogler. *Syntax-Directed Semantics -- Formal Models Based on Tree Transducers*. Springer Verlag, Berlin, 1998.
- [13] John Beck. *E-mail Explained* Available online <http://www.sendmail.org/email-explained.html>.
- [14] Thomas Perst. *fxmacro* Available online <http://www.informatik.uni-trier.de/~perst/fxmacro>.
- [2] John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, New York, second edition, 2001.
- [3] Oleg Kiselyov. *SXML as a higher-order markup language*, April 2001. Available online <http://okmij.org/ftp/Scheme/xml.html>.
- [4] Ralf Lämmel and Wolfgang Lohmann. Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*. OCG, 2001.
- [5] Erik Meijer and Mark Shields. XML: A Functional Language for Constructing and Manipulating XML Documents. In *Submitted to USENIX 2000 Technical Conference*, 2000.
- [6] Andreas Neumann. *Parsing and Querying XML Documents in SML*. PhD thesis, University Trier, 1999.

- [7] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, second edition, 2000.
  - [8] R. Milner, M. Tofte, R. Harper and D. MacQueen. *The definition of standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.
  - [9] Dan Suciu. Semistructured Data and XML. *Proceedings of International Conference on Foundations of Data Organization*, 1998.
- 

## The Authors

### Thomas Perst

*University Trier, Department of Computer Science*  
Trier  
54286  
Germany  
[perst@uni-trier.de](mailto:perst@uni-trier.de)

Thomas Perst is Ph.D. student at the University of Trier, Germany. He has graduated in 2001 from the Computer Science Department of the University of Trier, Germany.

### Helmut Seidl

*University Trier, Department of Computer Science*  
Trier  
54286  
Germany  
[seidl@psi.uni-trier.de](mailto:seidl@psi.uni-trier.de)

Helmut Seidl is a Professor with the University of Trier, Germany.

### Extreme Markup Languages 2002

Montréal, Québec, August 6-9, 2002

*This paper was formatted from XML source via XSL*

*Mulberry Technologies, Inc., August 2002*