DSD: A Schema Language for XML

Nils Klarlund
AT&T Labs–Research
klarlund@research.att.com

Anders Møller
BRICS, University of Aarhus
amoeller@brics.dk

Michael I. Schwartzbach
BRICS, University of Aarhus
mis@brics.dk

ABSTRACT

XML (eXtensible Markup Language) is a linear syntax for trees, which has gathered a remarkable amount of interest in industry. The acceptance of XML opens new venues for the application of formal methods such as specification of abstract syntax tree sets and tree transformations.

A notation for defining a set of XML trees is called a *schema language*. Such trees correspond to a specific user domain, such as XHTML, the class of XML documents that make sense as HTML.

A useful schema notation must: identify most of the syntactic requirements that the documents in the user domain follow; allow efficient parsing; be readable to the user; allow limited tree transformations corresponding to the insertion of defaults; be modular and extensible to support evolving classes of XML documents.

In the present paper, we introduce the DSD (Document Structure Description) notation as our bid on how to meet the requirements above.

The expressiveness of DSDs goes far beyond the DTD concept that is already build into XML and SGML. In particular, we advocate the use of nonterminals in a top-down manner, coupled with boolean logic and regular expressions to describe how constraints on tree nodes depend on their context. We also support a general, declarative mechanism for inserting default elements and attributes that is reminiscent of Cascading Style Sheets (CSS), a way of manipulating formatting instructions in HTML that is built into all modern browsers. Finally, we include a simple technique for evolving DSD documents through selective redefinitions.

The DSD language is completely self-describable, meaning that the syntax of legal DSD documents together with all static requirements are captured in a special DSD document, the meta-DSD of less than 500 lines.

We relate DSDs to other recent XML schema languages and to languages for abstract syntax description.

The DSD language is fully implemented and is available in an open source distribution.

1. INTRODUCTION

XML (eXtensible Markup Language)[5] is a syntax derived from SGML for markup of text. XML is particularly interesting to computer scientists because the markup notation is really nothing but a way of specifying labeled trees. The tree view and the convenient SGML syntax of HTML have been important to the development of the World Wide Web. Recently, the general XML syntax has gathered a remarkable amount of interest in industry as a way of exchanging data such as documents, databases, or computed information.

We argue in this document that the acceptance of XML opens new ways of introducing formal computer science techniques into practice. Specifically, we study: the formal specification of XML languages, that is sets of abstract syntax trees, and tree-based default insertion mechanisms, that is, tree transformations. Both aspects are part of the DSD (Document Structure Description) notation, which we introduce informally in this article. Before we explain DSDs, let us mention some fundamental XML efforts that are already standardized (in the sense of being a W3C recommendation) or under development:

- *CSS (Cascading Style Sheet)* allows XML documents to be rendered visually (CSS2[1] is the latest official recommendation);
- *transformation language* defines rather general transformations between XML languages (XSLT[6], which is also called a style sheet language, became an official recommendation recently);
- *linking* defines generalized links between XML resources (X Link[11] and XPointer[10] are almost completed, whereas XPath[7], a simple expression language underlying several of the XML efforts, has just been turned into an official recommendation);
- *schema language* is a current effort similar to ours for describing the formal syntax of XML applications (XML has already inherited the DTD concept from SGML, but this notation is considered inadequate by many); and
- *query language*, which will generalize database queries to semi-structured data represented by XML documents.

In the area of schema languages, several proposals, such as DDML[2], DCD[3], SOX[9], Schematron[14], and RELAX[19] have already emerged. Recently, W3C has issued an official draft proposal for XML Schema, which has been met with intense debate.

Our DSD proposal is more ambitious than other proposals with the exception of Schematron, which is based on a pattern matching paradigm instead of a parsing view, and RELAX, which is more expressible in some regards. A DSD defines a grammar for a class of XML documents, documentation for that class, and additionally a CSS-like notation for specifying default parts of documents. A DSD is itself an XML document.

We recall that an XML document consists of elements that have attributes and content; the latter is a sequence of text interspersed with subelements. For an example, take the HTML markup

```
<body class='mystuff'>
  Hello <em>there</em>
</body>
```

This is an element representing a node labeled “body”. The node has an attribute named “class” and two children corresponding to its content (the stuff between the start tag `<body..>` and the end tag `</body>`: a text node with value “Hello” and an element node labeled “em”; the “em” node in turn has one child node, which is a text node.

We have six major goals for the descriptive power of DSDs. They should:

- allow context dependent descriptions of content and attributes, since the context of a node, such as ancestors and attribute values, often govern what is legal syntax;
- generalize CSS[1] (Cascading Style Sheets) so that readable, CSS-like rules for default attribute values and default content can be defined for arbitrary XML domains, not only predefined user formatting models;
- complement XSLT[6] in the sense that the expressive power of DSDs should be close to that of XSLT, so that assumptions made by XSLT style sheets can be made explicit in a DSD;
- permit the description of semi-structured data, that is, the description of what references may point to;
- enable the redefinitions of syntactic classes, so that evolving XML languages can be expressed in terms of existing DSDs; and
- be self-describable.

It is also important to us that a DSD yields a linear time algorithm for checking conformance of XML documents and that DSDs are based on simple concepts familiar to computer scientists. To honor these ambitions, our design combines several elementary ideas: a uniform notion of *constraint* that captures the legality of attributes, attribute values, and content; conditional constraints guarded by *boolean expressions* that capture dependencies between attributes, attribute values, element contexts, and content; *nonterminals* in the form of element IDs that allow several different versions of an element to coexist; the concept of *projected content* that allows succinct descriptions of both ordered and unordered content; *regular expressions* to describe both attribute values and content sequences; automatic insertion of *default* attributes and elements guided by boolean expressions; several ID types to allow easy redefinitions; and *points-to* requirements that constrain the targets of references.

Despite its expressive power, the DSD language is simple enough that it can be rigorously defined in an 15 page English specification (excluding examples and introduction). This present paper describes the main ideas of the language and relates it to other XML schema language proposals.

Related work

There are currently two major W3C initiatives aiming at describing classes of XML documents.

The first initiative is called RDF (Resource Description Framework) Schema Specification. RDF is a generic notation for describing metadata, such as content ratings, user references, or content relationships. It is based on well-known concepts: named properties and entity-relationship diagrams. Thus an RDF description is a graph expressed in a generic notation. RDF schemas, in turn, declare properties and allowed relationships that constrain the shape of an RDF description. An RDF schema defines a number of domains, mappings among them, and classes, which may be related by subclass constraints. Thus, RDF schemas aim at describing data models, not XML syntax as such.

The second initiative is named XML Schemas. The requirements that a schema language should address are summarized in the document[17]. The DSD language, we believe, satisfies the principles and requirements outlined, except that we have paid less attention to a precise coordination with other W3C standards (some of which are under development). In particular, we have not addressed the relatively modest issue of integrating primitive datatypes with our structural descriptions. Neither have we addressed the issue of namespaces[4].

The XML Schema proposal[22] contains many features that may directly be compared to the DSD language. Other features, such as those that deal with name spaces and import mechanisms, are outside the scope of the current DSD proposal.

- The XML Schema proposal introduces several mechanisms, inspired by object-oriented programming, for restraining how schemas are constructed such as final, abstract, and equivalence notions. They contribute to the complexity of the language, while impeding self-describability. The current DSD proposal does not rely on object-orientation, since we found that many application domains, such as HTML, do not lend themselves to 00.
- In XML Schema, a number of constraint-like concepts are introduced: complex types, attribute group definition, and content type concepts. We propose to unify these, and our additional notion of a boolean constraint, into one concept.
- The XML Schema proposal introduces three different kinds of content models element content model, element-only content, and named model group. We introduce only one kind of content model, which may be anonymous or identified by an ID.

Apparently, the current XML Schema proposal does not satisfy two of the requirements in[17]:

- it is not self-describing, since many syntactic constraints on schemas are not describable by a schema; and
- it does not address schema evolution, that is, how existing schemas may be combined or amended to reflect new features or restrictions.

We address the first requirement by making the DSD language strong enough to cover boolean conditions, including context descriptions and the description of where ID references point to. Our meta-DSD, which describes the class of valid DSDs, covers all syntactic constraints.

We address the second requirement by our use of definitions and redefinitions of nonterminals as a simple solution to the problem of extending grammatical categories in schemas as they evolve.

There are other significant differences between DSDs and XML Schema. First, our notion that attributes must be declared gradually avoids semantic ambiguities in how CSS is used for inserting default attribute values for XML languages like SMIL[13]. Second, our schema language captures that content and attribute declarations often depend on ancestors and other attributes; XML Schema does not allow attributes value dependencies, which are common to XML languages (including XML Schema itself). Finally, the key notions of XML Schema of how to specify the recursive structure of a document are, in our opinion, too weak and at the same time much too complicated as far as we gather from the current draft[22].

There have been several other schema language proposals. DDML[2] was the result of a collaborative effort on the XML-DEV mailing list. It is a relatively straightforward generalization of DTD concepts. A similar notion called DCD was proposed in [3]. A different approach was suggested by SOX[9], which is based on an object-oriented paradigm. These languages do not appear to offer a unifying notion of constraint, or context-dependent declarations.

A interesting approach[21], called assertion grammars, achieves some of our goals since it is based implicitly on nonterminals. Recast in our terminology, assertions are redefinitions of nonterminals that conditionally extend their meaning. The condition reflects the context where the addition is valid. We believe it would be possible to explain assertion grammars fully in terms of DSD concepts; conceivably, assertion grammar concepts could be integrated with DSDs, where they would stand for abbreviations of DSD constructs. Assertion grammars allow only a restricted class of extensions, and they do not allow as flexible context dependencies as DSDs.

Another approach closely related to ours is that of RELAX[19], which is based on the automata-theoretic characterization of regular tree languages formulated in [18]. In RELAX, a specification expresses a nondeterministic tree automaton. In order to decide whether a given document is accepted by the automaton, an efficient algorithm must work bottom-up in order to carry out a subset construction on the fly. We depart fundamentally from RELAX on this point: we chose to make DSDs similar to deterministic, top-down automata—otherwise, it would not be obvious how DSDs could become a foundation for CSS extended to arbitrary XML. With our semantics, defaults are inserted deterministically as a part of the parsing process; had we chosen a more general automaton model, default insertion would become very complex—seemingly amounting to the solution of a kind of system of equations. Indeed, RELAX is suggested as a notation that is explicitly designed not to support default insertions. We disagree: declarative default mechanisms are so important that they must be supported by the semantics of the schema notation.

Our notion of constraint assignment is superficially similar to the way automata states are assigned by RELAX to nodes of the XML tree; also, we use the idea of[18] to express the transition relation by regular expressions over automata states. However, our current semantics is that of a parsing process, not that of automata theory. (We may in the future decide on a purer semantics, although we are committed to a top-down approach.)

We know of no other work that have suggested a generalization of CSS based on a schema notation; the Simple Tree Transformation Language outlined in[12] seems similar in ambition but is based on a more operational, and explicit, semantics.

The DSD notation is similar in some respects to the XSLT transformation language: both employ a top-down traversal of a tree based on testing properties, such as attribute values, of a current node and its ancestors. They differ in that the XSLT expression language is more powerful (so that more properties can be tested), the output may look very different from the input (whereas DSDs only insert element and attribute default), and that there are no unique named constraints assigned to nodes during parsing. DSD with its more restricted formal apparatus allow features such as CSS-like defaults, linear parsing, and redefinitions, which are hard to achieve with XSLT.

Similarly, the Schematron[14] proposal is not based on grammatical structures, but uses patterns expressed in XPath/XSLT to impose collections of individual requirements that could possibly be used in conjunction with grammar-based schema notations.

2. XML CONCEPTS

The reader is assumed familiar with standard XML concepts, such as those defined in [5]. The following provides a brief description of the XML object model used in DSDs.

A well-formed XML document is represented as a tree. The leaf nodes correspond to empty elements, chardata, processing instructions, and comments. The internal nodes correspond to non-empty elements. DTD information is not represented. Each element is label-led with a name and a set of attributes, each consisting of a name and a value. Names, values, and chardata are strings.

Child nodes are ordered. The *content* of an element is the sequence of its child nodes. The *context* of a node is the path of nodes from the root of the tree to the node itself. Element nodes are ordered: An element v is *before* an element w if the start tag of v occurs before the start tag of w in the usual textual representation of the XML tree.

Processing instructions with target `dsd` or `include`, as well as elements and attributes with namespace `http://www.brics.dk/DSD`, contain information relevant to the DSD processing. All other processing instructions and also chardata consisting of white-space only and comments are ignored.

3. THE DSD LANGUAGE

A DSD defines the syntax of a family of conforming XML documents. An *application document* is an XML document intended to conform to a given DSD. It is the job of a *DSD processor* to determine whether an application document is conforming or not.

A DSD is itself an XML document. This section describes the main aspects of the DSD language and its meaning. For a complete definition, we refer to [16].

A DSD is associated to an application document by placing a special processing instruction in the document prolog. This processing instruction has the form

<?dsd URI="*URI*"?>

where *URI* is the location of the DSD. A DSD processor basically performs one top-down traversal of the application document in order to check conformance. During this traversal, nodes are assigned *constraints* by the DSD. A constraint specifies a requirement of a node relative to its context and content that must be fulfilled in order for the document to conform. Initially, a constraint is assigned to the root node. During the checking of a node, its child nodes are assigned new constraints, which are checked later in the traversal. Also, checking a constraint may cause default attributes and child nodes to be inserted. The term *the current element* is used in the following to refer to the node in the application document that is being processed at a given moment during the traversal.

If no constraints have been violated upon termination, then the original document conforms to the DSD and the resulting document with defaults inserted is output.

A DSD consists of a number of definitions, each associated with an ID allowing reference for reuse and redefinition. In the following, the various kinds of definitions are described. We use a number of small examples, some inspired by the XHTML language[20] and some that are fragments of the book example described in Section 5.

3.1 Element constraints

The central kind of definition is the *element definition*. An element definition defines a pair consisting of an element name and a constraint. During the application document processing, the elements in the application documents are assigned IDs of such element definitions. An element can only be assigned the ID of an element definition of the same name.

The IDs of element definitions are reminiscent of nonterminals in context-free grammars. Each ID determines the requirements imposed on the content, attributes, and context of the element to which it is assigned. We allow several different element definitions with the same name; thus, element names are not used as nonterminals. This distinction allows several versions of an element to coexist.

As an example, consider a DSD describing a simple database containing information about books, such as, their titles, authors, ISBN numbers, and so on. Imagine that both the whole database and each book entry should contain a `title` element, but with different structure. Book entry titles may only contain chardata without markup; also, defaults may be specified for book entry titles. Database titles may contain arbitrary content and no attributes. These two kinds of `title` elements can be defined as follows:

```
<ElementDef ID="book-title" Name="title"
            Defaultable="yes">
  <Content><StringType/></Content>
</ElementDef>

<ElementDef ID="database-title" Name="title">
  <ZeroOrMore>
    <Union>
      <StringType/><AnyElement/>
    </Union>
  </ZeroOrMore>
</ElementDef>
```

A constraint is defined by a constraint expression, which can contain declarations of attributes, declarations of element content, boolean expressions about attributes and context, and conditional subconstraints guarded by boolean expressions. These aspects are described in the following sections.

The example below expresses something that is impossible or cumbersome to formalize in other schema proposals. The requirement is that anchor elements in XHTML are not nested:

```
<ElementDef ID="a">
  <Constraint>
    <Not>
      <Context>
        <Element Name="a"/><SomeElements/>
      </Context>
    </Not>
  </Constraint>
  ...
<ElementDef>
```

3.2 Attribute declarations

During evaluation of a constraint, attributes are declared gradually. Only attributes that have been declared are allowed in an element. Since constraints can be conditional and attributes are declared inside constraints, this evaluation scheme allows hierarchical structures of attributes to be defined. Such structures cannot be described by other schema proposals although they are common; for instance, in a XHTML `input` element, the `length` attribute may only be present if the `type` attribute is present and has value `text` or `password`.

An attribute declaration consists of a name and a string type. The name specifies the name of the attribute, and the string type specifies the set of its allowed values. It is an error if an attribute being declared is not present in the current element, unless it is declared as optional.

The presence and values of declared attributes can be tested in boolean expressions and context patterns. For instance:

```
<Attribute name="action">
  <StringType IDRef="URI"/>
</Attribute>
```

evaluates to *true* if an attribute named `action` has been declared, is present in the current element, and its value matches the string type `URI`.

Our notion of gradual attribute declaration is essential to the use of CSS-like mechanisms in generic XML settings. For example, the proposed use of CSS in SMIL[13] is not entirely well-defined: with a CSS-like mechanism both setting and testing attributes in no pre-defined order the result of default insertion is ambiguous. (This ambiguity does not appear when CSS is used to set formating properties that live in a different universe from attributes.)

3.3 String types

A *string type* is a set of strings defined by a regular expression. String types are used for two purposes: to define valid attribute values and to define valid chardata.

Regular expressions provide a simple, well-known, and expressive formalism for specification of sets of strings. All reasonable sets can be defined, and by the correspondence with finite-state automata, an efficient implementation is possible. A rich set of operators is provided, such as `Sequence`, `ZeroOrMore`, `Union`, `Optional`, `Intersection`, `Complement`.

The use of regular expressions is more flexible than using a predefined collection of data types. Furthermore, the relations-ship to finite-state automata guarantees an efficient implementation. Special automata representations, such as MONA[15], promises that this approach extends to Unicode[8].

All well-known data types, such as URIs, e-mail addresses, and ZIP codes, can be described by regular expressions. The following example shows the definition of ISBN numbers:

```
<StringTypeDef ID="isbn">
  <Sequence>
    <Repeat Value="9">
       <Sequence>
         <CharSet Value="0123456789"/>
         <Optional>
           <CharSet Value=" -"/>
         </Optional>
       </Sequence>
    </Repeat>
    <CharSet Value="0123456789X"/>
  </Sequence>
</StringTypeDef>
```

3.4 Content expressions

The content of an element, its child nodes, can be viewed as a sequence of element nodes and chardata nodes. We ignore other kinds of nodes and assume that there are no adjacent chardata nodes. (Adjacent ones may be joined by concatenation.)

As a part of an element constraint, a set of valid content sequences can be specified using a formalism which resembles regular expressions, but is modified to take default insertion into account.

Content expressions are built of atomic expressions and content expression operators. An atomic expression is either an element description or a string type. Element descriptions are used to assign constraints to the child elements, and string types specify chardata child nodes. There is no backtracking across constraint assignments to child elements: once a sequence of children has been matched, the assignment of constraints to them is fixed, and parsing continues in a top-down manner.

The operators consist of `Sequence`, `ZeroOrMore`, `AnyElement`, `Union`, `If`, and a few others.

As an example, the valid content of a XHTML `table` element (see [20], App. A.1) can be described by the following content expression:

```
<Sequence>
  <Optional>
    <Element IDRef="caption"/>
  </Optional>
  <Union>
    <ZeroOrMore>
      <Element IDRef="thead"/>
    </ZeroOrMore>
    <ZeroOrMore>
      <Element IDRef="tfoot"/>
    </ZeroOrMore>
  </Union>
  <Optional>
    <Element IDRef="thead"/>
  </Optional>
  <Optional>
    <Element IDRef="tfoot"/>
  </Optional>
  <Union>
    <OneOrMore>
      <Element IDRef="tbody"/>
    </OneOrMore>
    <OneOrMore>
      <Element IDRef="tr"/>
    </OneOrMore>
  </Union>
</Sequence>
```

Modulo the syntactic overhead of the XML notation, this example could just as easily be expressed in DTD. But, as explained in the following, DSDs also allow more complex content requirements to be specified.

A constraint may contain a collection of content expressions. Each of them must match some of the content of the current element, just like each attribute declaration must match an attribute. More precisely, each content expression is matched against a subsequence of the content that consists of elements mentioned in the content expression itself. Thus, the actual content is *projected* onto the elements that the content expression is about. If, for instance, the content expression mentions elements `A` and `B`, and the content is a sequence of elements `A`, `B`, `C`, a chardata node, and an element `A`, then this expression is matched against the projected content `A`, `B`, `A` (and the match fails). This method makes it easy to specify requirements of both *ordered* and *unordered* content. Additionally, unordered content is declared just like attributes.

In the XHTML specification, the content of the `head` element is described as "`head.misc`, combined with a single `title` and an optional `base` element in any order". In a DTD, this requirement can be formalized only by listing all the possible combinations in a single regular expression. The XML schema proposal introduces a separate operator to express interleavings. In a DSD, three content expressions in a constraint does the job:

```
<Content IDRef="head.misc"/>
<Element IDRef="title"/>
<Optional><Element IDRef="base"/></Optional>
```

When checking a set of content expressions, each of them are thus checked in turn on their own subsequence of the content. As an final requirement, each content node must be matched by exactly one content expression. Thus, generally speaking, content expressions in a constraint must not overlap, just as it is an error to declare an attribute more than once.

3.5 Context patterns

A context pattern can be used to make defaults, constraints and content descriptions context dependent.

Context patterns are defined in essence like CSS selectors[1]. A context pattern is a sequence of context terms; a context term is either an *element pattern* or a `SomeElements` element. The context of the current element is a sequence of nodes, starting at the root of the XML tree, and ending in the current element. The context of the current element matches a context pattern if the context can be decomposed into consecutive fragments, such that the sequence of context terms match the sequence of fragments. An element pattern specifies an element name and a set of attributes, and is matched by a single element node if the name and attributes match. A `SomeElements` is matched by any context fragment. Implicitly, all context patterns begin with a `SomeElements` element.

The following example is a context pattern that matches those `li` elements that are immediately within `ul` elements inside `form` elements whose `method` attribute has value `post`:

```
<Context>
  <Element Name="form">
    <Attribute Name="method" Value="post"/>
  </Element>
  <SomeElements/>
  <Element Name="ul"/>
  <Element Name="li"/>
</Context>
```

To see how useful context-dependent definitions are, let us consider a common situation: an XML grammar that represents not one but several related XML notations. For example, a DSD may specify both draft and final markup notations for books. This is the scenario mentioned in the XML 1.0 specification, where conditional sections of DTDs may be used to describe variations:

```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >
<![%draft;[
<!ELEMENT book (comments*, title, body,
                supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>
```

Here, two flags (macros or parameter entities), called `draft` and `final` are used to control the expansion of the two conditional definitions of `book`. Typically, these flags would be declared in the document type declaration of the application document, whereas the conditional sections would be declared in an external DTD. The declarations in the application document are processed before the external DTD.

As stated, the first conditional definition is expanded since the first item of the conditional definition expands to `INCLUDE`. Similarly, the second definition is not expanded since the first item expands to `IGNORE`. In our opinion, this mechanism is cumbersome and unsafe. A document writer must set two flags at the same time, and they must not both be `INCLUDE` or `IGNORE`.

With DSDs, the parameterization of the XML grammar can be explained in terms of the application document itself. For example, if the root element is called `DOC`, then an attribute `draft` of this element would govern the definition of a `book`:

```
<ElementDef ID="book">
  <Sequence>
    <If>
      <Context>
        <Element Name="DOC">
          <Attribute Name="draft"
                     Value="true"/>
        </Element><SomeElements/>
      </Context>
      <Then><ZeroOrmore>
        <Element IDRef="comments"/>
      </ZeroOrMore></Then>
    </If>
    <Element IDRef="title"/>
    <Element IDRef="body"/>
    <Optional>
      <Element IDRef="supplements"/>
      </Optional>
  </Sequence>
</ElementDef>
```

Here the logic of the different versions is clearly spelled out at the XML level of the application document itself. We believe that this simple mechanism is not possible with any other of the XML schema proposals.

3.6 Default insertion

Default attributes and content are defined by an association to a boolean expression. Such attributes or content is *applicable* for insertion at a given place in the application document if the boolean expression evaluates to true at that place.

The following example defines that the `length` of `input` fields of type `text` is by default 20:

```
<Default>
  <Context>
    <Element Name="input">
      <Attribute Name="type" Value="text"/>
    </Element>
  </Context>
  <DefaultAttribute Name="length" Value="20"/>
</Default>
```

Defaults are inserted "upon request" by constraints:

- When an attribute declaration is encountered and the declared attribute is not present in the current element, an applicable default is inserted, if a such exists.
- During evaluation of a content expression, if an element description or a string type is encountered and the next content node does not match the description, then an applicable default is inserted, if a such exists. Default elements can only be inserted if declared as defaultable by the description.

A notion of *specificity* of defaults, based on CSS[1], is used to determine a default when more than one is applicable. Intuitively, the default with the most complex boolean expression is chosen; if two are equally complex, the one latest defined is chosen.

For convenience, defaults can also be defined in the application document. Every application document element may contain default definitions, which in a sense extend the DSD. Such default definitions are recognized using the DSD namespace. They are not considered part of the application document by the DSD processor. Their scope are not the whole application document; they are considered as applicable default definitions only in the subtree rooted by the element in which they occur.

The following example shows how the length default previously defined may be overridden for certain text type input elements, namely those inside form elements that have an action attribute whose value is a string starting with http://www.brics.dk/:

```
<DSD:Default>
  <Context>
    <Element Name="form">
      <Attribute Name="action"/>
        <Sequence>
          <String Value="http://www.brics.dk/"/>
          <ZeroOrMore><AnyChar/></ZeroOrMore>
        </Sequence>
      </Attribute>
    </Element>
    <SomeElements/>
    <Element Name="input">
      <Attribute Name="type" Value="text"/>
    </Element>
  </Context>
  <DefaultAttribute Name="length" Value="30"/>
</DSD:Default>
```

Defaults defined in the application document are always considered more specific than defaults defined in the DSD document. Furthermore, when two application document defaults are applicable and they are not siblings, the one with the smallest scope, that is, the inner-most, will always be considered more specific than the other.

Our notion of default mechanism goes much beyond other schema proposals. We do believe that CSS is so useful and well-established that a generic version should be adopted along with a schema language.

3.7 ID attributes and points-to requirements

In attribute declarations, a DSD may declare that application document attributes are of type ID or IDRef, as is also possible with DTDs. An attribute of type ID is considered a *definition* of the value of the attribute. Such a definition must be unique. Similarly, an IDRef attribute is a *reference* to the element containing the attribute defining the given value, and such an element must exist.

Additionally, a DSD may impose a *points-to* requirement on the element denoted by a reference. Such a requirement is defined by a boolean expression, which may probe attribute values and context as we have seen. This mechanism allows description of semi-structured data.

In the following example, a book-reference attribute is declared. It must refer to an element with an attribute of type ID occurring in a book element:

```
<AttributeDecl ID="book-reference"
               IDType="IDRef">
  <PointsTo>
    <Context><Element Name="book"/></Context>
  </PointsTo>
</AttributeDecl>
```

Points-to requirements are checked in a separate phase after the main traversal of the XML tree.

As explained in Section 4, the DSD language is self-describable. The meta-DSD relies on the ID mechanism to enforce proper use of definitions.

3.8 Redefinitions and evolving DSDs

It is often the case that a whole class of related XML schemas is to be defined. Also, often one wants to create an XML schema from an existing schema by making modifications and extensions. DSDs support these software practices by providing two simple mechanisms: *document inclusion* and *redefinition*.

Both DSD documents and application documents can be created as extensions of other documents using a special include processing instruction of the form

```
<?include URI="URI"?>
```

where *URI* denotes the document to be included, that is, inserted in place of the processing instruction. A document can only be included once into a given document; subsequent attempts are ignored.

In DSDs, all definitions can be renewed. One can include a document containing a definition of a concept and then later redefine the concept. Since the DSD language is designed to be self-describable, the meta-DSD must be able to express this notion of redefinition.

In order to allow modification and extension of existing application document definitions, two new attribute types, *RenewID* and *CurrIDRef*, are introduced beside ID and IDRef. All definitions can be redefined using RenewID; an IDRef attribute refers to the *last* occurring definition or redefinition in the document. An attribute of type CurrIDRef refers to the *current definition*, which is the last definition or redefinition occurring before which does not contain the element with the CurrIDRef attribute. Assume that in some existing DSD, a book element has been defined as follows:

```
<ElementDef ID="book">
  <Constraint IDRef="book-constraints"/>
</ElementDef>

<ConstraintDef ID="book-constraints">
  ...
</ConstraintDef>
```

Consider a situation where we want to reuse this DSD but would like to extend the book constraints with a new attribute declaration. This can be done using RenewID to redefine book-constraint and CurrIDRef to refer to the original definition:

```
<ConstraintDef RenewID="book-constraints">
  <Constraint CurrIDRef="book-constraints"/>
  <AttributeDecl Name="new-attribute"/>
</ConstraintDef>
```

3.9 Self-documentation

Documentation may be associated to most constructs in a DSD. This is treated as meta-information, which does not affect the processing. It allows a DSD to be virtually self-documenting towards application authors. Also, a DSD processor may use this information when errors are detected to provide the author with useful help.

The DSD language allows three kinds of documentation: `Label`, which can be used to attach a label to the construct; `Doc`, which is intended for full documentation of the construct; and `BriefDoc`, intended for a brief description, which could be translated in a title attribute of HTML (the effect is that a box with the brief documentation pops up when the mouse is over the construct). Documentation may consist of arbitrary XML, but a XHTML-like subset is recommended.

4. THE META-DSD

The DSD language is self-describable: there is a DSD that completely captures the requirements for an XML document to be a valid DSD. We provide such a DSD of less than 500 lines, called the *meta-DSD*. It can be used both as a human readable description of DSD to clarify unclear issues, and by DSD processors to check whether a given XML document is a valid DSD. The meta-DSD resides at `http://www.brics.dk/DSD/dsd.dsd`; thus, all DSD documents should contain the processing instruction:

```
<?dsd URI="http://www.brics.dk/DSD/dsd.dsd"?>
```

stating that they are intended to conform to the meta-DSD.

5. THE BOOK EXAMPLE

We now present a small example of a complete DSD. It describes an XML syntax for databases of books. Such a description could be arbitrarily detailed; we have settled for title, ISBN number, authors (with homepages), publisher (with homepage), publication year, and reviews. The main structure of the DSD is as follows:

```
<?dsd URI="http://www.brics.dk/DSD/dsd.dsd"?>

<DSD IDRef="database" DSDVersion="1.0">
  <ElementDef ID="database">
    <ZeroOrMore>
      <Element IDRef="book"/>
    </ZeroOrMore>
    <Element IDRef="database-title"/>
  </ElementDef>
  ...
</DSD>
```

In the `database` element we use projected content to allow the `title` to appear anywhere. The remaining definitions are presented below, excluding the `title` element and the `isbn` string type that are shown in Section 3.

```
<ElementDef ID="book">
  <AttributeDecl Name="isbn" Optional="yes">
     <StringType IDRef="isbn"/>
  </AttributeDecl>
  <Sequence>
    <If><Attribute Name="isbn"/>
        <Then>
          <Optional>
            <Element IDRef="book-title"/>
          </Optional>
        </Then>
        <Else>
          <Element IDRef="book-title"/>
        </Else>
    </If>
    <OneOrMore>
      <Element IDRef="author"/>
    </OneOrMore>
    <Element IDRef="publisher"/>
    <Element Name="year">
      <StringType IDRef="digits"/>
    </Element>
    <Optional>
      <Element Name="review">
        <StringType IDRef="url"/>
      </Element>
    </Optional>
  </Sequence>
</ElementDef>
```

The `isbn` attribute is optional; if it is not present in a `book`, then a `title` is mandatory.

```
<ElementDef ID="author">
  <Sequence>
     <Element Name="first">
       <StringType IDRef="simple"/>
     </Element>
     <Optional>
       <Element Name="initial">
         <StringType IDRef="simple"/>
       </Element>
     </Optional>
     <Element Name="last">
       <StringType IDRef="simple"/>
     </Element>
  </Sequence>
  <Optional>
    <Element IDRef="homepage"/>
  </Optional>
</ElementDef>

<ElementDef ID="publisher">
  <StringType IDRef="simple"/>
  <Optional>
    <Element IDRef="homepage"/>
  </Optional>
</ElementDef>
```

An order is imposed on `first`, `initial`, and `last`, but projected content allows the optional `homepage` element to appear anywhere.

```
<ElementDef ID="homepage">
  <StringType IDRef="url"/>
</ElementDef>
```

```
  <StringTypeDef ID="url">
    <ZeroOrMore><AnyChar/></ZeroOrMore>
  </StringTypeDef>
```

A naive definition of `url` is chosen here. It could be replaced with the full 200 line official definition, which is indeed a regular language.

```
  <StringTypeDef ID="simple">
    <OneOrMore>
      <Union>
        <CharRange Start="a" End="z"/>
        <CharRange Start="A" End="Z"/>
        <CharSet Value="._- &amp;"/>
      </Union>
    </OneOrMore>
  </StringTypeDef>

  <StringTypeDef ID="digits">
    <ZeroOrMore>
      <CharRange Start="0" End="9"/>
    </ZeroOrMore>
  </StringTypeDef>
```

Such string types should be part of a standard library.

```
  <Default>
    <Context>
      <Element Name="book"/>
    </Context>
    <DefaultContent>
      <title>Untitled</title>
    </DefaultContent>
  </Default>
```

This definition allows untitled books to receive the default title `Untitled`. An example of a conforming application document looks as follows:

```
<?dsd URI="http://www.brics.dk/DSD/book.dsd"?>

<database>
  <title>
    <b>Classic Computer Science Books</b>
  </title>
  <book isbn="0201485419">
    <title>
      The Art of Computer Programming
    </title>
    <author>
      <first>Donald</first>
      <initial>E</initial>
      <last>Knuth</last>
      <homepage>
        http://www-cs-faculty.stanford.edu/
                                    ~knuth/
      </homepage>
    </author>
    <publisher>
      Addison-Wesley
      <homepage>http://www.aw.com</homepage>
    </publisher>
    <year>1998</year>
    <review>
      http://www.amazon.com/exec/obidos/ASIN/
                                 0201485419
    </review>
  </book>
</database>
```

6. THE XPML EXAMPLE

At AT&T Labs, we have used DSDs to describe XPML, an HTML-like experimental language that has been developed for programming IVR (Interactive Voice Response) systems. The XPML notation has evolved from being a simple version of HTML, dubbed PML, to becoming a rich programming environment for telephone services that rely on text-to-speech, touchtone input, speech recognition, and call control.

Often, XPML documents resemble conventional marked-up documents; but sometimes they are heavily customized with many default time and prompt settings, making them more like notations in a programming language. DSDs play an important role, since they can describe almost all syntactic constraints of the language. (Indeed, the needs of PML originally motivated the development of the DSD language.)

The XPML core: big picture

XPML has a simple core, similar to HTML; for example, `statements` comprise `select`, `a` and `menu` elements and `inline` content, where `inline` is text or `audio` or `span` elements. This part of the syntax is describable by DTDs as well. The DSD reflects the fact that the syntax really is more complicated: `form` statements may occur only when not nested inside another `form` statement, and `input` statements may occur only inside a `form` statement. These context dependencies are easily expressed using a combination of boolean logic and regular expressions.

The XPML core: attribute dependencies

The `type` attribute of the `input` statement determine what other attributes are possible and what the allowed content is. For example, when the `type` attribute is `text`, a `size` attribute is allowed.

Platform specific markup

Variations in hardware or device choices influence language constructs, such as attributes and their value ranges. These constraints are modeled in separate DSDs that amend the description of the core XPML language. For example, the `xpml-att` DSD describes metric attributes for controlling how information about user sessions are reported back to the server.

Additional abstractions

At the other end of the abstraction spectrum is the need for numerous variations on basic constructs, such as the select element. Each variation is a generic interaction style characterized by how the user is prompted and how error situations are handled. Each interaction style is itself further parameterized by various messages, timeout parameters, and so on. These variations would be hard describe using other formal techniques such as object-oriented types; they are simply too heterogeneous. Nevertheless, they look rather much alike on the surface. As an example, the `menu` element is already described in the core DSD as containing elements according a regular expression

```
<ElementDef ID="menu">
  <OneOrMore>
    <Sequence>
      <Element IDRef="menu-option"/>
      <Optional>
        <Element IDRef="menu-do"/>
      </Optional>
    </Sequence>
  </OneOrMore>
  <Constraint IDRef="menu-constraint"/>
  <Constraint IDRef="menu-dtmf-constraint"/>
</ElementDef>
```

The content expression denotes any non-empty sequence of `option` elements, where each `option` element allows an optional `do` element immediately following it. Also, a constraint `menu-constraint` is introduced to be a place holder for attributes common to both speech and touchtone (DTMF) input as the DSD evolves; similarly, `menu-dtmf-constraint` is introduced as a place holder for touchtone specific constraints.

In a separate DSD that describe the interaction style abstraction, the `interaction` attribute that selects an interaction style, either `basic` or `optional`, is introduced, along with the extra elements `counttimeout` and `pause` that are allowed:

```
<ConstraintDef RenewID="menu-constraint">
  <Constraint CurrIDRef="menu-constraint"/>
  <AttributeDecl Name="interaction"
                 Optional="yes">
    <StringType IDRef="Menu-interaction-name"/>
  </AttributeDecl>
  <If>
    <Or>
      <Attribute Name="interaction"
                 Value="basic"/>
      <Attribute Name="interaction"
                 Value="optional"/>
    </Or>
    <Then>
      <Element Name="counttimeout"
               Defaultable="yes">
        <Constraint IDRef=
                    "message-attributes"/>
        <Content IDRef=
                 "menu-message-content"/>
      </Element>
      <Element Name="pause" Defaultable="yes">
        <Constraint IDRef=
                    "message-attributes"/>
        <Content IDRef="menu-message-content"/>
      </Element>
    </Then>
  </If>
</ConstraintDef>
```

Platform dependent defaults

The number of defaults for XPML is very large; there are many patterns in the assignments, and the CSS-like default mechanism is particularly suited for capturing the defaults in as systematic a way as possible. For example, we can express that both `select` and `menu` elements share certain default values for some of their attributes:

```
<Default>
  <Or>
    <Context><Element Name="select"/></Context>
    <Context><Element Name="menu"/></Context>
    <Context><Element Name="input"/></Context>
  </Or>
  <DefaultAttribute Name="maxtterrs"
                    Value="3"/>
  <DefaultAttribute Name="maxmisselected"
                    Value="3"/>
  <DefaultAttribute Name="maxtimeout"
                    Value="2"/>
  <DefaultAttribute Name="endchars"
                    Value="#"/>
  <DefaultAttribute Name="interdigittimeout"
                    Value="4000ms"/>
  <DefaultAttribute Name="finaltimeout"
                    Value="5000ms"/>
  <DefaultAttribute Name="timeout"
                    Value="0ms"/>
</Default>
```

We have made a preliminary description of the full XPML language. Our experiments show that almost all of the syntax and static semantics of XPML can be captured as DSDs.

7. THE DSD 1.0 TOOL

A prototype DSD processor has been implemented and is freely available. This prototype shows that it is possible to implement a complete DSD processor in less than 5000 lines of simple C code. The processor tests conformance of application documents and inserts defaults.

By using a DSD processor as a front-end for other XML tools, these often become much simpler to construct. The DSD processor itself relies on this technique. Using the meta-DSD it can be checked that a given DSD document is indeed a valid DSD. Then, when using this DSD to check the actual application documents, the processor simply assumes that the DSD is valid. This bootstrapping technique has reduced the size of the implementation and made it more readable.

The DSD language is designed such that a linear-time processing is possible. The prototype fulfills this property; execution time is proportional to the size of the application document (where DSD defaults are viewed as belonging to an extended DSD). The constant of proportionality depends on the complexity of the given DSD. As an example, a self-application of the meta-DSD takes less than half a second.

Functionality

The DSD tool is given the URI of an application document containing a DSD-reference processing instruction. It performs the traversal of the application document as described in Section 3, and if it succeeds, it then performs the points-to check described in Section 3.7.

Before the application document is processed, the DSD document (including all application document defaults) is checked to see whether it conforms to the meta-DSD. This check can be omitted by a command-line option if the user is certain that the DSD is in fact valid.

If an error occurs, that is, if a document is not conforming to its DSD, then a suitable error message is inserted in the document which is then output. If the processing succeeds without errors, then the defaults are added to the application document. As an extra feature, the tool can be instructed to add special attributes that detail the element ID assigned to a node. Such parsing information can be useful in subsequent processing by other XML tools.

Availability

The DSD processor is available in an open source distribution. Please visit the DSD project home page at: `http://www.brics.dk/DSD/` for more information. This home page also contains other DSD resources, such as the official specification of the DSD 1.0 language, example DSDs and application documents, an XSL style-sheet[6] allowing a pleasing visual rendering of DSD documents, and more.

8. CONCLUSION

The DSD language provides a simple but very expressive alternative to other XML schema proposals. It embodies a formal approach to the specification, validation, and default completion of XML syntax. It addresses issues such as context dependencies, CSS-like defaults, schema evolution, semi-structured data, complex data types, and efficient implementation. It has an expressive power similar to XSLT in the sense that XSLT recursion (even with modes) and testing based on boolean expressions probing element and attribute content is expressible as DSDs. Moreover, the DSD language has been implemented and tested in practice (and the implementation is freely available).

9. REFERENCES

[1] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs, editors. *Cascading Style Sheets, level 2, CSS2 Specification*. W3C, 1998.
URL: `http://www.w3.org/TR/REC-CSS2/`.

[2] Ronald Bourret, John Cowan, Ingo Macherius, and Simon St. Laurent, editors. *Document Definition Markup Language (DDML) Specification, Version 1.0*. W3C, 1999.
URL: `http://www.w3.org/TR/NOTE-ddml`.

[3] Tim Bray, Charles Frankston, and Ashok Malhotra, editors. *Document Content Description for XML*. W3C, 1998.
URL: `http://www.w3.org/TR/NOTE-dcd`.

[4] Tim Bray, Dave Hollander, and Andrew Layman, editors. *Namespaces in XML*. W3C, 1999.
URL: `http://www.w3.org/TR/REC-xml-names`.

[5] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0*. W3C, 1998.
URL: `http://www.w3.org/TR/REC-xml`.

[6] James Clark. *XSL Transformations (XSLT) Specification*. 1999. URL: `http://www.w3.org/TR/WD-xslt`.

[7] James Clark and Steve DeRose, editors. *XML Path Language*. W3C, 1999.
URL: `http://www.w3.org/TR/xpath`.

[8] The Unicode Consortium. *The Unicode Standard, Version 2.0*. Addison Wesley, 1996.
URL: `http://www.unicode.org/`.

[9] A. Davidson et al. *Schema for Object-Oriented XML 2.0*. W3C, 1999.
URL: `http://www.w3.org/TR/NOTE-SOX/`.

[10] Steve DeRose, Ron Daniel Jr., and Eve Maler, editors. *XML Pointer Language*. W3C, 1999.
URL: `http://www.w3.org/TR/xptr`.

[11] Steve DeRose, Eve Maler, David Orchard, and Ben Trafford, editors. *XML Linking Language*. W3C, 2000.
URL: `http://www.w3.org/TR/xlink`.

[12] Daniel Glazman. Simple tree transformation sheets 3. Technical Report NOTE-STTS3-19981111, W3C, 1998. http://www.w3.org/TR/NOTE-STTS3.

[13] Philipp Hoschka et al. *Synchronized Multimedia Integration Language (SMIL) 1.0 Specification*. W3C, 1998.
URL: `http://www.w3.org/TR/REC-smil`.

[14] Jeff Jelliffe, editor. *The Schematron: An XML Structure Validation Language using Patterns in Trees*. 1999.
URL: `http://www.ascc.net/xml/resource schematron/schematron.html`.

[15] Nils Klarlund and Anders Møller. *MONA Version 1.3 User Manual*. BRICS, 1998.
URL: `http://www.brics.dk/mona`.

[16] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. *Document Structure Description 1.0*. AT&T & BRICS, October 1999. URL: `http://www.brics.dk/DSD/specification.html`.

[17] Ashok Malhotra and Murray Maloney. *XML Schema Requirements*. W3C, 1999.
URL: `http://www.w3.org/TR/NOTE-xml-schema-req`.

[18] Makoto Murata. Hedge automata: a formal model for xml schemata, 1999. `http://www.xml.gr.jp/relax/hedge_nice.html`.

[19] Makoto Murata. How to relax. Technical report, xml.gr, 2000. `http://www.xml.gr.jp/relax/`.

[20] Steven Pemberton et al. *XHTML 1.0: The Extensible HyperText Markup Language*. W3C, 1999.
URL: `http://www.w3.org/TR/WD-html-in-xml`.

[21] Dave Raggett. Assertion grammars. Draft,
URL: `http://www.w3.org/People/Raggett/dtdgen/Docs/`, 1999.

[22] Henry S. Thompson et al. *XML Schema Part 1: Structures*. 2000.
URL: `http://www.w3.org/TR/xmlschema-1/`.