The DSD Schema Language and its Applications

Nils Klarlund
Anders Møller
Michael I. Schwartzbach

BRICS Report Series RS-00-41

ISSN 0909-0878 December 2000

The DSD Schema Language and its Applications[*]

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

Anders Møller & Michael I. Schwartzbach
BRICS, University of Aarhus
{`amoeller,mis`}`@brics.dk`

Abstract

XML (eXtensible Markup Language), a linear syntax for trees, 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 user domain may be specified as a set of trees. For example, XHTML is a user domain corresponding to the set of XML documents that make sense as HTML. A notation for defining such a set of XML trees is called a *schema language*. We believe that 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 a declarative default notation à la CSS; and be modular and extensible to support evolving classes of XML documents.

In the present paper, we give a tutorial introduction to the DSD (Document Structure Description) notation as our bid on how to meet these requirements. The DSD notation was inspired by industrial needs, and we show how DSDs help manage aspects of complex XML software through a case study about interactive voice response systems (automated telephone answering systems, where input is through the telephone keypad or speech recognition).

The expressiveness of DSDs goes beyond the DTD schema concept that is already part of XML. 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 DSDs through selective redefinitions. DSDs are in many ways much more expressive than XML Schema (the schema language proposed by the W3C), but their syntactic and semantic definition in English is only 1/8th the size. Also, the DSD notation is self-describable: the syntax of legal DSD documents and *all* static semantic requirements can be captured in a DSD document, called the *meta-DSD*.

[*]This article is a revised version of "DSD: A Schema Language for XML" [18]; in addition, material from [15] has been included.

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. Thus, it may not be surprising that XML syntax has been hyped as a universal solution to the pervasive problem of format incompatibility.

Such generous promises notwithstanding, at least one fascinating and fundamental quality sets XML-based notations apart from ad hoc syntax: they encourage tree transformations—a technique that application programmers usually do not take advantage of. In fact, it would probably be considered a hassle even to define a set of parse trees and procedures according to which they are constructed and parsed. XML circumvents this problem by offering a primary representation based on trees, at the expense of syntactic succinctness. Of course, trees and mappings between trees are a main ingredient of computer science; for example, such mappings are essential to building compilers, where the compilation process is partitioned into several phases, most of which simply massage one intermediate tree format into another one.

The purpose of the present article is to indicate how XML opens new ways of applying formal computer science techniques to general, practical problems. Specifically, we study the formal specification of XML languages, that is sets of abstract syntax trees, and tree-based default insertion mechanisms for common tree transformations needed by application programmers. 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 technologies that are already standardized (in the sense of being a W3C recommendation) or under development:

- *CSS (Cascading Style Sheet)*: for documents to be rendered visually through (1) a simple tree transformation language and (2) a target language of text properties for layout. CSS2 [1] is the latest official recommendation.

- *Transformation language*: for rather general transformations between XML languages. XSLT [6], which is also called a style sheet language, became an official recommendation in 1999.

- *Linking*: for generalized links between XML resources. XLink [11] and XPointer [10] are almost completed, whereas XPath [7], a simple expression language underlying several of the XML efforts, is already an official recommendation.

- *Schema language*: 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). The W3C notation is called XML Schema [25] and it has recently achieved Candidate Recommendation status.

- *Query language*: for generalizing database queries to semi-structured data represented by XML documents.

- *Namespaces*: for allowing an XML document to consist of syntax from several domains at once; typically, an XML document may be extended with syntax that is foreign to its primary use—such syntax must be specially tagged. Namespaces are introduced into XML in [4], but their meaning has been the subject of controversy.

In the area of schema languages, several proposals, such as DDML [2], DCD [3], SOX [9], Schematron [14], and RELAX [22] have already emerged. Recently, W3C has issued an official draft proposal for XML Schema, which has been met with intense debate.

Our DSD proposal—which is rigorously summarized in [17]—is more ambitious than other proposals, perhaps 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 (at least in an earlier version, where it allowed all regular tree languages to be expressed). 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 named *elements* representing tree nodes. Elements have *attributes*, representing name/value pairs, and *content*, which is text (called *chardata*), interspersed with subelements. For an example, take the HTML markup:

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

This text is an element named "`body`" that corresponds to a tree 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 the DSD notation. These goals are by no means comprehensive, of course. But they reflect most of the needs we have seen in document processing and database applications. The only major omission is the concept of namespaces, whose semantics until recently have been the subject of uncertainty. DSDs 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 15 pages [17] (where the page count excludes examples and introduction). The specification of the Structural Part of XML Schema runs to about 140 pages (counted in the same way)—although it must be admitted that XML Schema does address the important area of namespaces. The present paper describes the main ideas of the DSD notation and relates it to other XML schema language proposals. We also provide an account of an industrial example that motivated DSDs: HTML-like languages for defining Interactive Voice Response systems (that is, user interfaces that work through spoken prompts and telephone pad or speech input).

In the rest of the article

After an overview of the XML tree model in Section 2, we introduce most DSD concepts through little examples in Section 3, and we explain the concept of meta-DSD. In Section 4, we present a complete DSD for book data, and, in Section 5, we discuss how an application programmer would benefit from DSDs when learning and using a domain specific language for IVR (Interactive Voice Response) applications. We describe our prototype implementation of the DSD processor in Section 6. In Section 7, we discuss related work, in particular XML Schema and RELAX. We give a conclusion in Section 8.

2 XML Concepts

The reader is assumed familiar with the most common XML concepts (XML is officially defined in [5]). We now give 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 (text), processing instructions, and comments. The internal nodes correspond to non-empty elements. For that reason, we often abuse language by confounding elements and nodes. DTD information is not represented. Each element is labeled 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 according to *document order*: 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. The observant reader may have noticed that our representation allows adjacent text nodes; in fact, we will assume that trees are a normalized by a process that matches such nodes by concatenation of their text.

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 [17].

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, constraints and other requirements from the DSD are *evaluated* relative to a *current element (node)* of the application document. The DSD processor consults the DSD to determine the constraints that are *assigned* to each node for later evaluation. Initially, a constraint is assigned to the root node. Evaluation of a constraint may entail the insertion of default attribute values and default content in the current element.

If no constraints have been violated during the traversal, then the original document conforms to the DSD. The document augmented with inserted defaults is the result of the DSD processing.

A DSD consists of a number of definitions, each associated with an ID so that they can be referred to and, possibly, redefined. In the following, the various kinds of DSD definitions are described. We use a number of small examples, some inspired by the XHTML language [23] and some that are fragments of the book example described in Section 4.

3.1 Element constraints

The definition central to DSDs is the *element definition*. An element definition specifies an element name and a constraint. During conformance checking, each node of the application document is assigned an ID of an element definition. Naturally, an element can be assigned only the ID of an element definition for the same name as that of the element.

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 must contain a `title` element, but with different structure. Book entry titles may contain only chardata (and no 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 an XHTML `input` element, the `length` attribute may be present only 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, the expression

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

evaluates to *true* if and only if the attribute named `action` satisfies two conditions: it has been declared and it is present in the current element with a value matching 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 formatting 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. Many 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`, and `Complement`.

The use of regular expressions is more flexible than using a predefined collection of data types. Furthermore, the relationship to finite-state automata guarantees an efficient implementation. Special automata representations for large alphabets, such as MONA [16], holds the promise 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

Recall that the content of an element, its children, is a sequence of element nodes and chardata nodes. *Content expressions* are used to specify sets of such sequences. These expressions are a kind of regular expression; they occur in element constraints.

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 element children, and string types specify chardata child nodes. There is no non-local backtracking across constraint assignments to children: once a sequence of children has been matched for a given element, the assignment of constraints to them is fixed, and parsing continues in a top-down manner. (Backtracking, however, is possible for a sequence of children as long as not all of them have been matched.)

The content expression operators include `Sequence`, `ZeroOrMore`, `AnyElement`, `Union` and `If`.

As an example, the valid content of a XHTML `table` element (see [23], 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. With a DSD, a set of three content expressions in a constraint does the job:

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

When such a set of content expressions is evaluated, each of them is evaluated on *projected content*, namely the subsequence of the content that mentions the element names in the expression. Additionally, each content node must be matched by exactly one content expression. Thus, generally speaking, content expressions in a constraint must not overlap with respect to element names they mention, just as it is an error to declare an attribute more than once.

3.5 Context patterns

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

Context patterns are very similar to CSS selectors [1]. A context pattern is a sequence of context terms; a *context term* is either an element pattern or a `SomeElements` element. An *element pattern* specifies an element name and a set of attributes. 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.

Before summarizing the meaning of context patterns, we provide an example of a context pattern that matches those `li` elements 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>
```

The matching semantics of contexts is as follows. The context of the current element is matched by a context pattern if the context can be decomposed into consecutive fragments such that the sequence of context terms matches the sequence of context terms in the pattern. An element pattern matches a single element node if the name and attributes match (in the obvious way). A `SomeElements` matches any context fragment. Implicitly, all context patterns begin with a `SomeElements` element.

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 somewhat 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 (perhaps with the exception of Schematron).

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 be inserted only 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 is 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 the prefix `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. Moreover, when two application document defaults are applicable and they are not siblings, the one with the smallest scope, that is, the innermost one, will always be considered more specific than the other.

In Section 5, we will look at examples that involves managing a great number of interdependent defaults.

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 the description of semi-structured data (such as the DSD notation itself, see Section 3.10.).

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 application document.

3.8 Redefinitions and evolving DSDs

In practice, not one but a whole class of related XML schemas is to be defined. In particular, an XML schema is often created from an existing schema through 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.

To accommodate modifications of DSD 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 *final* definition or redefinition in the document for that `ID`. An attribute of type `CurrIDRef` refers to the *current definition*, which is the last definition or redefinition occurring before the reference (and that does not contain it). 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. Documentation 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.

3.10 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 (allowing sometimes several tags on the same line), 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.

4 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 home pages), publisher (with home page), 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>
```

5 Industrial case study

IVR (Interactive Voice Response) systems range from simple telephony applications ("press 1 for sales, press 2 for customer service") to complicated dialogue systems based on speech recognition. But even the simpler systems are notoriously difficult to construct since their programming involves timing and error issues that always tend to get complex. To simplify the task, many layers of abstractions are introduced. At the highest level, an application programmer is then mainly concerned about choosing pre-canned dialogues, which are filled in with a variety of parameters, such as prompts and timeout durations. In this section, we will study how XML and the DSD Schema may help an application programmer learn and use a specialized notation with many interdependent parameters such as prompts, timeout values, error counts, error messages, etc. In particular, we will show how a DSD processor automates the filling-in of defaults for such parameters according to the programmer's preferences.

Our case study is based on XPML (Extensible Phone Markup Language), an HTML-like experimental language developed at AT&T Labs. The XPML notation has evolved from being a simple version of HTML, dubbed PML, to becoming a rather elaborate programming notation 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. For such markup language applications, DSDs may play an important role, since they can describe almost all syntactic constraints, while providing a practical solution to the handling of defaults. (Indeed, the needs of PML originally motivated the development of the DSD language.)

(The XPML notation as outlined here is somewhat incomplete. It is similar to VoiceXML, a new dialogue markup language developed by AT&T, IBM, Lucent and Motorola. VoiceXML is not very similar to HTML, but otherwise resembles XPML in scope and purpose.)

5.1 The IVR scenario

We will present our case study from the application programmer's point of view. Our scenario calls for this programmer to develop a little whimsical, interactive voice application that probes the mood of a customer. The programmer will use XPML (for Extensible Phone Markup Language), which he is not yet very familiar with. The main idea of PML is that simple HTML-like pages describe a finite-state machine, where intra-page hyperlinks become goto statements and text becomes synthesized speech; input fields corresponds to subdialogues for obtaining numbers and select elements become dialogues à la “for sales, choose 1; for customer service, choose 2,...”.

Each subdialogue construct provides numerous parameters for specifying prompts, help messages, timeout durations, timeout counts, and messages in various error situations. As a further complication, there are several interdependencies among these parameters. For example, some HTML elements are associated with several possible *interaction styles* that support situations such as: unusually many choices in a menu, number input restricted to certain ranges, variations in dialogue style (“press any key when you hear the right choice”), etc. The interaction style is specified by an `interaction` attribute. Naturally, the kinds of prompt parameters, along with many other settings, are dependent on the value of this attribute.

5.2 DSDs for syntax explanations

Our application programmer wants to use XPML, but he doesn’t know much about it except for some examples he has seen. Naturally, he will mainly use these examples for guidance, but the DSD may provide a readable, concise syntactic summary. We do not envisage that the programmer will read the DSD as an XML file. Instead, a hyperlinked HTML document may be produced by an XSLT style sheet transformation. For example, the DSD definition of the element `XPML`, the top element of an XPML document, is shown below (left) through an XSLT style sheet transformation into HTML. The pretty-printed version is designed to resemble the concrete syntax of an application document; the original DSD definition (right) probably should not be shown to the application programmer:

<XPML> ID=*XPML*:
(**<head>**
Constraint *head-constraint*
</head> [Defaultable],
<body>
Constraint *body-constraint*
</body>)
</XPML>

```
<ElementDef ID="XPML">
  <Sequence>
    <BriefDoc>
      The head element
      may be omitted.
    </BriefDoc>
    <Element Name="head"
            Defaultable="yes">
      <Constraint IDRef=
        "head-constraint"/>
    </Element>
    <BriefDoc>
      The body element is
      mandatory.
    </BriefDoc>
    <Element Name="body">
      <Constraint IDRef=
        "body-constraint"/>
    </Element>
  </Sequence>
</ElementDef>
```

The `BriefDoc` documentation strings of the XML version are translated into HTML `title` attributes—they provide the effect of a pop-up explanation when the mouse pointer is over the corresponding definition. This particular snippet of a DSD specifies that the `XPML` element consists of a `head` element followed by a `body` element. The `head` is defaultable (which means that it may be omitted if a default for it has been specified), and its attributes and content are specified by the constraint named `head-constraint`. Similarly, the `body` element is specified by the constraint `body-constraint`. The XSLT style sheet can be found at the DSD Web site (at `http://www.brics.dk/DSD`); it is rather complicated, approximately 25 pages.

5.3 DSDs for debugging

Now, we will explore how schemas may help debug XML documents. Let's assume that the application programmer's first attempt at the mood-probing XPML program is:

```
<?dsd URI="xpml-att.dsd"?>
<XPML>
  <head>
    <application name="HELLOWORLD"/>
    <maintainer address="karam@research.att.com"
                loglevel="2"/>
    <title>The Greeting Application</title>
  </head>
  <body>
    Welcome to greetings are us.
    <span nointerrupt="y">
      <audio url="/audioclips/greeting.vox"/>
    </span>
    <a name="repeat"/>
    <menu name="feelings">
      <option dtmf="0">To end</option>
      <do><a href="#endit"/>
         <comment>go to end point</comment>
      </do>
      <option> If you are feeling like a cowboy. </option>
      <do> Howdy world! </do>
      <option> If you are feeling like a Canadian. </option>
      <do> Gid’day world, how’s it going eh? </do>
    </menu>
    <a href="#repeat"/>
    <a name="endit"/>
  </body>
</XPML>
```

The programmer has inserted a `<?dsd URI="xpml-att.dsd"?>` processing instruction to mark that the document must conform to the DSD named `xpml-att.dsd`. He can now use the DSD processor to check the syntax of the document. It’ll tell him:

```
Error in ’greetings-first-attempt.pml’
line 10: attribute ’nointerrupt’ has illegal value ’y’
while checking attribute in constraint
"message-attributes", ’xpml-core.dsd’ line 377
```

An automated error analysis tool would display this constraint along with pertinent auxiliary definitions:

ConstraintDef ID=*message-attributes*:
  nointerrupt=”*YesOrNo*”[Optional]

StringTypeDef ID=*YesOrNo*:
  (”*yes*” | ”*Yes*” | ”*no*” | ”*No*”)

So, the programmer must write `"yes"`, not `"y"`. Naturally, the other schema notations offer similar capabilities. Most people will probably get acquainted with schemas only through such error-reporting (grammar reading being not a favorite pastime of programmers)—thus, it is very important that the schema notation itself is a simple as possible, otherwise error messages will be difficult to interpret for the non-expert.

5.4 DSDs for myriads of defaults

Once the above error is corrected, the DSD processor accepts the document and inserts all the default attributes and default elements specified by the DSD for XPML. The resulting document is:

```
<?dsd URI="xpml-att.dsd"?>
<XPML>
  <head>
    <application name="HELLOWORLD"/>
    <maintainer address="karam@research.att.com" loglevel="2"/>
    <title>The Greeting Application</title>
  </head>
  <body>
    Welcome to greetings are us.
    <span nointerrupt="yes">
       <audio url="/audioclips/greeting.vox"/>
    </span>
    <a name="repeat"/>
    <menu asrmode="none" endchars="#" finaltimeout="5000ms"
          interaction="basic" interdigittimeout="4000ms"
          maxmisselected="3" maxtimeout="2" maxtterrs="3"
          name="feelings" timeout="0ms">
      <option dtmf="0">To end</option>
      <do><a href="endit"/><comment>go to end point</comment></do>
      <option> If you are feeling like a cowboy. </option>
      <do> Howdy world! </do>
      <option> If you are feeling like a Canadian. </option>
      <do> Gid'day world, how's it going eh? </do>
      <help>No help is available.</help>
      <initial>
        <enumerate><option/>Press
          <emph><dtmf/></emph>.
        </enumerate>
      </initial>
      <timeout> You have exceeded the time limit. </timeout>
      <toomanyerrors> Sorry, too many errors. </toomanyerrors>
      <counttimeout> Sorry, too many timeouts. </counttimeout>
      <pause> Pausing.  Press pound sign to continue. </pause>
    </menu>
    <a href="repeat"/>
    <a name="endit"/>
  </body>
</XPML>
```

It is similar to the original document except that all timing and counting parameters that are relevant according to the schema have been inserted. Also, various default messages used in error and help situations, like `<help>No help is available. </help>` have been inserted. Voice programming, as well as layout via HTML layout, is dependent on a great number of parameters whose tuning is often essential to obtaining the right performance. But just as with HTML, they are not usually something that the programmer wants to explain in detail for every part of the document, since it would ruin the document as an abstract representation of the contents.

This example shows how DSDs generally allow XML notations to be abstracted away from rendering details in a way similar to CSS. However, we should note that DSDs do not quite subsume CSS: in the domain of visual formatting, there are some arithmetic rules about inheritance of values that cannot currently be expressed in DSDs.

DSD style sheets

DSD defaults defined by both the system and the application programmer may be gathered in files known as *external parsed entities*. These are just like XML documents except that multiple root elements are allowed. They work as style sheets by inclusion in the application document via the `include` processing instruction.

Below, the application programmer has defined a DSD style sheet that overrides the default `help` element for the `menu` construct in two ways: for a `menu` without a `class` attribute, the message "We're sorry, can't help you more right now, but please call us at 1-800-greetings" is specified; for a `menu` with a `class` attribute of value `moody`, the default content of `help` becomes "How are you feeling, dud? press 1 to get relief",

```
<DSD:Default>
  <Context>
    <Element Name="menu"/>
  </Context>
    <DefaultContent>
    <help>
      We're sorry, can't help you more right now,
      but please call us at 1-800-greetings
    </help>
  </DefaultContent>
</DSD:Default>

<DSD:Default>
  <Context>
    <Element Name="menu">
      <Attribute Name="class" Value="moody"/>
    </Element>
  </Context>
  <DefaultContent>
     <help>
       How are you feeling, dud?  press 1 to get relief
     </help>
  </DefaultContent>
</DSD:Default>
```

Thus, parameters can be gathered hierarchically in files to achieve the cascading effect that enable abstractions, formulated as sets of defaults, to be easily further customized.

5.5 The XPML core: big picture

We will now describe how a description of XPML may be layered. (The collection of DSDs mentioned in the following can be found at `http://www.brics.dk/DSD`.) XPML has a simple core, similar to HTML; for example, a DSD (and even a DTD) may easily express that `statements` comprise `select`, `a` and `menu` elements and `inline` content, where `inline` is text or `audio` or `span` elements. However, 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>
```

5.6 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>
```

5.7 DSDs for simplifying XPML processing

With a DSD processor, an XML documents may be *normalized* by default insertion in the sense that (1) without inserted defaults (assuming all default information is erased from the DSD) the document is not conforming and (2) with defaults inserted the document is conforming and no more defaults would be inserted—if it were to be run again. (Strictly speaking, DSD semantics currently present the problem that default element insertion may cause the DSD processing to be a non-idempotent operation, so the preceding statement is not always true when one expects it to be true.)

Since defaults cannot be removed by defaults in an application document, only overridden, the defaults given with the DSD itself provide a set of assumptions about the shape of the document that results from running the DSD processor on a valid document. For example, the XPML interpreter can assume `menu` elements are fully filled-in with timing attributes and content such as `help` and `error` messages, since an application programmer provided default can change this information, but not let it disappear.

For this reason, the system programmer, who is writing a semantic interpreter for XPML, may omit a host of error and default situations that would otherwise be typical of a domain specific language like XPML. In other words, the DSD notation itself, with its emphasis on parameters and defaults, becomes a domain modeling tool that directly simplifies the building of software.

5.8 Summary of DSD advantages

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. We have illustrated four practical aspects of DSD schemas:

- DSDs aid the XPML programmer to choose the right syntactic constructs. DSDs are by themselves not easy too read because of the XML syntax, so we indicate how to present them in a more conventional BNF-like way that closely resembles the concrete syntax of the XPML notation.
- XPML programmers can easily check their documents for most errors using the DSD processor alone.
- XPML programmers can use the CSS-like default mechanism that comes with DSDs. Thus, XPML programs can be "styled" in a declarative and modular fashion.
- DSD descriptions significantly simplify the programming of an interpreter for XPML.

In contrast, the XML Schema notation proposed by the W3C covers only the first two points, and only partly so: first, the notation is incapable of capturing much of the attribute structure of XPML, and second, the notation itself is so complicated (as argued previously) that it may impede its use as an explanatory medium directed towards computer professionals.

6 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, which is a complete description of DSDs, the processor checks that a purported DSD document is indeed a DSD. This bootstrapping technique has reduced the size of the implementation and made it more readable.

The DSD processor analyzes application documents in linear time in the following sense: 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. If DSD defaults in the application document are not discounted, then the running time is quadratic in the size of the document. In practice, the number of defaults is sufficiently small that the quadratic running time is not observed.

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, the XSLT style sheet for DSDs, and more.

7 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 [19]. 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 [25] contains many features that may directly be compared to the DSD language. Other features, such as those that deal with namespaces 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 this paradigm.

- 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 [19]:

- it is not self-describing, since many syntactic constraints on schemas are so complicated that they 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], where CSS is transplanted from a translator into a visual formatting language to a translator into XML. 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 [25].

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 [24], 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 [22], which is based on the automata-theoretic characterization of regular tree languages formulated in [20]. According to the original RELAX concept, 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 [20] 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.)

Recently, it was announced [21] that the RELAX project, influenced by the DSD notation, would adopt a top-down approach based on an automata-theoretic semantics.

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] is also meant to be easily understandable, but it 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. But the XSLT language is much more powerful (so that more properties can be tested—in fact, it is Turing-complete), the output may look very different from the input (whereas DSDs only insert element and attribute default), and there are no uniquely named constraints assigned to nodes during parsing. In practice, many XSLT programs visit each node only once; in this case, even the mode concept of XSLT has a counterpart in DSDs, namely IDs. DSDs with their more restricted formal apparatus allow features such as CSS-like defaults, linear parsing, and redefinitions, which are hard to achieve with XSLT.

In fact, 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. Being based on XSLT, Schematron is unlikely to be readable by the occasional XML programmer or application engineer.

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 reminiscent of XSLT since some XSLT recursion 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. It is our hope that DSD ideas may further simple

XML standards that go beyond just being grammar notations.

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*. W3C, 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. 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. From the programmer's point of view: XML for IVR and how DSD Schemas may help. Unpublished revision of "XPML: industrial case study", currently available at `http://www.research.att.com/projects/DSD/industrial-case/`.

[16] Nils Klarlund and Anders Møller. *MONA Version 1.3 User Manual*. BRICS Notes Series NS-98-3 (2nd revision), 1998. URL: `http://www.brics.dk/mona`.

[17] 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`.

[18] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. DSD: A schema language for XML. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice, FMSP'00*, 2000.

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

[20] Makoto Murata. Hedge automata: a formal model for XML schemata, 1999. `http://www.xml.gr.jp/relax/hedge␣nice.html`.

[21] Makoto Murata. Announcement on `http://www.xmlhack.com`, 2000.

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

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

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

[25] Henry S. Thompson et al. *XML Schema Part 1: Structures*. W3C, 2000. URL: `http://www.w3.org/TR/xmlschema-1/`.