The DSD Schema Language*

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 a set of XML documents that make sense as hypertext. 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 present in the user domains, and yet be sufficiently simple and easy to understand both by the schema authors and the users. Furthermore, it must allow efficient parsing and be modular and extensible to support reuse and evolution of descriptions.

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. We show how DSDs help manage aspects of complex XML software through a case study about interactive voice response systems, i.e., 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. Also, we include a simple technique for reusing and evolving DSDs through selective redefinitions. The expressiveness of DSD is comparable to that of the schema language XML Schema proposed by W3C, but their syntactic and semantic definition is significantly larger and more complex. Also, the DSD notation is self-describable: the syntax of legal DSD documents including all static semantic requirements can be expressed within the DSD language itself.

*This article is a revised version of [23]; in addition, material from [21] has been included.

1 Introduction

XML (Extensible Markup Language) [9] 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 since its introduction in 1998 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 transform one intermediate tree format into another one. XML has been suggested as an underlying notation for structuring and manipulating information in general. As a foundation of this, XML schemas are needed to formalize the sets of parse trees that constitute the individual languages.

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 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:

- *Syntax*: Schemas describe the formal syntax of XML languages. As for other formal languages, a precise syntax description provides an essential basis, both for the tool builders and the application users. XML has inherited the DTD schema concept from SGML, but this notation is considered inadequate by many. The newest schema notation from W3C is called XML Schema [34] and it has recently achieved Recommendation status. However, as explained later, this language is in our opinion not satisfactory, and several alternatives have been proposed.

- *Transformation*: Since XML encourages construction of highly specialized languages, there is a strong need for domain-specific languages that allow general transformations between XML languages to be defined more easily than possible with general-purpose programming languages. XSLT [10], the transformation part of the XSL language, became an official recommendation in 1999 and has become very popular.

- *Style sheets*: CSS (Cascading Style Sheets) is an example of a specialized transformation language, designed to make visual rendering for XML (and HTML) documents, which is a typical kind of transformation. It consists of a simple tree transformation language and a target language of text properties for layout. CSS2 [4] is the latest official recommendation.

- *Database querying*: Since XML documents in a sense generalize the relational database model to general semi-structured, there is a need for corresponding generalizations of query languages. A draft specification of the XQuery language [3] has recently been published. The term *schema* originates from the database community where it denotes descriptions of the structure of relations.

- *Linking and addressing*: XML is designed to operate on the Web, so notations for defining links between documents and for addressing fragments of documents are essential. XLink [17] allows generalized links between XML resources to be defined. It is based on XPointer [16], which in turn uses XPath [12] for expressing locations in XML documents in a robust manner. XPath is also used in XML Schema to express uniqueness constraints, in XSLT as a pattern matching mechanism, and in XQuery to express basic queries.

- *Namespaces*: XML languages are often built on top of other XML languages. This introduces the demand for a name space mechanism to be able to distinguish the various parts of an XML document. XML Namespaces [8] allows URIs to be associated with XML markup to be able to uniquely determine which sublanguages the markup belongs to. We mention namespaces here because they have implications to essentially all other XML technologies, in particular schema languages.

For a more thorough introduction to these concepts, we refer to [27]. Agreeing on well-designed languages for these fundamental technologies allows generic tools to solve problems common to many XML application languages. Agreeing on a simple but powerful schema language has the additional benefit of making it easier to design and learn new XML languages. In the area of programming languages, the BNF notation is an example of this phenomenon. Unarguably, the simplicity of that notation has been a requisite for its widespread use.

In the area of schema languages, numerous proposals, such as DDML [6], DCD [7], XML-Data [24], XDR [18], SOX [15], TREX [11], Schematron [20], Assertion Grammars [32], and RELAX [30], have already emerged. Recently, W3C has issued their XML Schema proposal [34] in an attempt to reconcile the efforts. However, it has been met with intense debate, primarily due to its unprecedented complexity viewed by many as being unnecessary and harmful [33, 1]. Concurrently, RELAX NG [13] has been developed as a descendant of RELAX and TREX and is now being standardized by OASIS. The many proposals, and the outcome of the XML Schema effort, indicate that it is far from obvious how the right schema language should be designed. In general, the XML notation turns out to be so versatile that it is hard to satisfy all design requirements and capture the various usage patterns, and at the same time keep the schema notation simple. We give a more thorough comparison between DSD and the most significant alternatives in Section 7.

Our DSD proposal—which is rigorously defined in [22]—has the ambition of providing an expressive power comparable to that of XML Schema and RELAX NG, while at the same time remaining simple to learn and use. We have tried to identify the most central aspects of XML language syntax and turn these into a clean set of schema constructs, based on well-known computer science concepts, such as boolean logic and regular expressions. A DSD defines a grammar for a class of XML documents, including documentation for that class, and additionally a CSS-like notation for specifying default parts of documents. As most other schema language proposals, the DSD language itself uses the XML notation. This opens up for the possibility of being self-describing, that is, having a DSD description of the DSD language.

We recall that an XML document consists of named *elements* representing tree nodes. Elements have *attributes*, representing name/value pairs, and *content*, which is Unicode text called *chardata*, interspersed with subelements. We here ignore comments and DTD information, and we assume that entity references have been expanded. For example, consider the following XHTML document fragment:

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

This fragment contains 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, that is, the part between the start tag `<body...>` and the end tag `</body>`. The first child is a text node with value `Hello`, and the other is an element node labeled `em`. The `em` node in turn has one child node, which is a text node. The markup is required to be *well-formed*, meaning that the begin and end tags are balanced and nested properly, which allows us to view XML documents as tree structures. A schema for XHTML would for example state that `class` attributes in fact are allowed in `body` elements, that chardata is allowed in the content, but also that for instance `body` elements cannot appear within the `em` tags. A schema language should make it possible to easily express such constraints.

Besides basing the DSD design on simple concepts that are familiar to computer scientists, we have a number of more technical goals for the descriptive power of the DSD notation. These goals are by no means comprehensive, but they reflect most of the needs we have seen in document processing and database applications:

- DSD should allow context dependent descriptions of content and attributes, since the context of a node, such as ancestors and attribute values, often governs what is legal syntax.
- Default attribute values and content should be defined in a declarative manner, separate from the structural descriptions. Thus we seek a generalization of CSS so that defaultable properties in the form of attributes and element content can be defined for arbitrary XML domains. CSS manipulates defaults, but only for properties in predefined formatting models.
- As most other schema languages, DSD should support node IDs and references for expressing non-tree-structured data. In addition, it should permit the description of what references may point to.

- In order to support development and maintenance of large schemas, DSD should contains mechanisms for schema evolution and reuse.
- DSD should be self-describable. This property allows schemas themselves to be viewed as application documents.
- The content model should be flexible enough to allow ordered and unordered content to be mixed.
- It should be possible to intersperse informal documentation with the formal language of schemas. That allows them to serve as complete language descriptions.
- Validity of chardata and attribute values should be defined with an extensible mechanism so that only a minimal number of primitive types are included in the core language.
- DSD should complement XSLT in the sense that assumptions made by XSLT style sheets about the shape of input documents can be made explicit.
- Finally, it is also important to us that a DSD yields a linear-time algorithm for checking conformance of XML documents.

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 element content guided by boolean expressions; a simple notion of redefinition combined with a schema inclusion mechanism for supporting extension and modularity; and *points-to* requirements that constrain the targets of references.

The only major omission is the concept of namespaces, whose semantics until recently has been the subject of controversy [5]. In the current version of DSD, we do apply namespaces within the DSD language, but we do not support namespaces in the application languages. We plan to add proper support for namespaces in a future version to mend this limitation.

Naturally, there are constraints that within reason can be conceived but are not expressible in our formalism. Moving to Turing complete formalisms would complicate the language unnecessarily. As in programming language grammar formalisms, it is customary to supplement a grammatical check with a few specialized routines written in a general programming language.

Despite its expressive power, the DSD language is simple enough that it can be rigorously defined in 15 pages [22] (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). 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 DSD: HTML-like languages

for defining Interactive Voice Response (IVR) systems, which are user interfaces that work through spoken prompts and telephone pad or speech input.

The main contribution of this work is the attempt to simplify and yet generalize existing XML schema languages. Also, we believe to have identified some essential design requirements and show that in particular boolean logic and regular expressions are useful formalisms in schema languages.

Outline

After an overview of the XML tree model in Section 2, we introduce the DSD concepts through little examples in Section 3, and we explain the notion of a meta-DSD. In Section 4, we present a complete DSD example for information about books. In Section 5, we describe a prototype implementation of the DSD processor, and in Section 6, we discuss how an application programmer would benefit from DSDs when learning and using a domain specific language for IVR applications. In Section 7, we discuss related work, in particular XML Schema and RELAX NG. We conclude in Section 8 with a summary of our experiences with DSD, followed by plans and ideas for future development.

2 XML Concepts

The reader is assumed familiar with the most common XML concepts (XML is officially defined in [9]). However, since there unfortunately is no common agreement on the terminology, we now give a brief description of the XML data model used in DSD.

A well-formed XML document is represented as a tree. The leaves correspond to empty elements, chardata, processing instructions, and comments. The internal nodes correspond to non-empty elements. For that reason, we often confound the terms "element" and "node". DTD information is not represented in the tree. Each element is labeled with a name and a set of attributes, which each consists of a name and a value. Names, values, and chardata are Unicode strings [14].

Child nodes are ordered. The *content* of an element is the sequence of its immediate 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 a is *before* an element b if the start tag of a occurs before the start tag of b in the usual textual representation of the XML tree. We will assume that trees are a normalized by a process that combines adjacent text nodes by concatenating 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 or not an application document is conforming. 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 [22].

A DSD can be 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. By inserting this in the application document, the author states that the document is intended to conform to the designated DSD.

A DSD processor basically performs one top-down traversal of the application document tree in order to check conformance. During this traversal, constraints and other requirements from the DSD are *evaluated* relative to a *current element* 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. Also, it may assign constraints to the subelements of the current element. If no constraints are violated during the entire tree traversal, the original document conforms to the DSD. The document augmented with inserted defaults constitutes the result of the DSD processing.

A DSD consists of a number of definitions, each associated with an ID for reference. In the following, the various kinds of DSD definitions are described. We use a number of small examples, some inspired by the XHTML language [31] and some that are fragments of the book example described in Section 4.

3.1 Element constraints

The central definition in DSD is the *element definition*. An element definition specifies an element name and a *constraint*. During conformance checking, each element node in the application document is assigned an ID referring an element definition from the DSD. In order for the element node to match the element definition, they must have the same name, and the element node must satisfy the constraint.

The IDs of element definitions are reminiscent of nonterminals in context-free grammars. Each ID determines the syntactic requirements imposed on the content, attributes, and context of the elements to which it is assigned. We distinguish between definition IDs and element names in order to allow several versions of an element to coexist. Thus, several different element definitions may occur with the same name. To avoid confusion about the term "ID", note that element definition IDs are references into the DSD and that multiple application document elements may be assigned the same ID.

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 structures. Book entry titles may contain only chardata and no markup, and defaults may be specified for them. Database titles may on the other hand contain arbitrary content and no attributes, and cannot be given by defaults. 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 number of constraint expressions, which can contain declarations of attributes and element content, boolean expressions about attributes and context, and conditional subconstraints guarded by boolean expressions. The constraint is satisfied if the evaluation of each constituent succeeds. These aspects are described in the following sections.

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

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

This element definition contains a single constraint expression, which is a simple boolean expression querying the element context. Note that the name attribute of the `ElementDef` is missing here. That simply means that the name is the same as the ID.

In DTD, the anchor nesting restriction cannot be formalized and merely appears as a comment. The DTD does exclude `a` elements from appearing immediately below other `a` elements, but, for instance, it allows `<a><b><a>...</a></b></a>`. Most other schema languages, including XML Schema, has the same limitation.

Boolean expressions are build from the usual boolean operators, `And`, `Or`, `Not`, `Imply`, etc., and are used for several purposes: they express dependencies between attributes, and they are used as guards in conditional constraints and default declarations, as explained later.

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`. In most schema language, this kind of constrains are not expressible. Their solution is to allow all combinations and resort to other means, typically general programming languages, for expressing the extra requirements. However, since dependencies are a very common phenomenon in XML languages, this is clearly not satisfactory. Another typical example can be found in the XML Schema specification [34], Section 3.2.3: "`default` and `fixed` may not both be present [...] if `default` and `use` are both present, use must have the actual value `optional` [...] if `ref` is present, then all of `<simpleType>`, `form` and `type` must be absent". Surprisingly, even though the XML Schema language repeatedly uses such dependencies itself, they cannot be expressed in XML Schema. In contrast, the conditional constraints and boolean expressions in DSD capture this notion of dependencies in a straightforward manner.

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. Unless it is declared as optional, an attribute must be present if it is declared. Conversely, only declared attributes are allowed to be present.

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`.

The CSS language can assign properties to the elements in a document, based on context-sensitive selectors. In generic XML settings where properties appear as element attributes, such as in SMIL [19], this can lead to semantic ambiguities since setting and testing of attributes occurs in no pre-defined order. Our notion of gradual attribute declaration avoids such ambiguities.

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. Special automata representations for large alphabets hold the promise that the efficient regular expression implementations extend to Unicode[1].

[1] See e.g. `http://www.brics.dk/automaton/`

Most well-known data types, such as URIs, email 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>
         <CharRange Start="0" End="9"/>
         <Optional><CharSet Value=" -"/></Optional>
       </Sequence>
    </Repeat>
    <CharSet Value="0123456789X"/>
  </Sequence>
</StringTypeDef>
```

This defines ISBN numbers to consist of 10 digits, optionally separated by single blanks or dashes, and where the final digit may also be the character 'X'. In a more familiar notation, this regular expression would be written as `([0-9][ -]?){9}[0-9X]`. The benefit of our more voluminous notation is that the syntactic structure of the expression is immediate from the XML structure.

In comparison, other schema languages typically provide a number of predefined data types and focus less on flexibility and user defined types. More details are given in Section 7.

3.4 Content expressions

Recall that the content of an element 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 that 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. An element description is essentially a reference to an element definition. It matches a given element node if their names match. The string types specify chardata child nodes. Checking that content sequences satisfy the given constraints has the side-effect that element definition IDs are assigned to the subelements. Also, as explained in Section 3.6, insertion of default content occurs while checking content expressions. Because of these side-effects, we need a non-standard interpretation of the regular expression constructs occurring in content expressions, in order to get a well-defined behavior.

The content expression operators include `Sequence`, `ZeroOrMore`, `AnyElement`, `Union` and `If`. A `Sequence` is matched with a content expression by a left-to-right traversal. For `ZeroOrMore`, the traversal is eager, that is, in continues as long as there is a match of the subexpression. For `Union`, the traversal allows backtracking. Each option is tried, and the first one that matches is chosen. The `If` construct defines a conditional subexpressions.

As an example, the valid content of a XHTML `table` element (see [31], 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>
```

Ignoring 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 more than one content expression. Each of them then 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 contains. If, for instance, a content expression mentions elements `A` and `B`, and the content is a sequence of elements `A`, `B`, `C`, followed by a chardata node and an element `A`, then this expression is matched against the projected content `A`, `B`, `A`. This method makes it easy to combine 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, however, it cannot be combined arbitrarily with the other content description operators. With DSD, a simple constraint with three content expressions 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 the *projected content*, namely the subsequence of the content that mentions the element names in the expression. The first expression only looks at the elements that occur in `head.misc` (which is defined elsewhere); the second only looks at `title` elements and states that there must be exactly one of these; and the third expression states that there can be an optional `base` element somewhere in the content. 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. This simple and intuitive approach is unique to DSD.

For another example, consider the combination of the following two content expressions:

```
<Sequence>
  <Element IDRef="first"/>
  <Element IDRef="initial"/>
  <Element IDRef="last"/>
</Sequence>
<Optional><Element IDRef="homepage"/></Optional>
```

Together they require that the content consists of the three elements `first`, `initial`, and `last` occurring in that order, and that a single `homepage` element may optionally occur anywhere in that sequence. Without multiple content expressions and the notion of projected content, all possible combinations would have to be be explicitly listed.

As explained in Section 7, the content description mechanisms in other schema languages are typically also based on variations of regular expressions, in some cases adding a notion of inheritance. The solution to the problem of expressing combined ordered and unordered content is however unique to DSD.

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 [4]. 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. Recall that we define the *context* of the current element to be the sequence of nodes that start at the root of the XML tree and end 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 fragments matches the sequence of context terms in the pattern. An element pattern matches a single element node if the name and attributes match. 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 (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`. This mechanism is somewhat unsafe: a document writer must set two flags at the same time, and their values must be opposite each other.

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 expressing this logic is not possible with any of the other schema language proposals, since they do not have equivalent notions of context expressions and conditional constraints. One exception is Schematron, which employs the powerful language XPath for expressing constraints, as explained in Section 7.

3.6 Default insertion

It is convenient to application document authors to be able to omit implied attributes and other document parts. Since schemas describe the document structure, they are a suitable place to specify default values for such parts. Validating a document then has the side-effect of inserting the defaults, which is often useful to subsequence document processing.

In DSD, 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.

In other schema languages, the most common approach is to specify the defaults together with the structural descriptions, for instance at the attribute declarations. However, by specifying the defaults separately in a declarative manner, the default mechanism becomes more flexible because it allows variations of the default values. The IVR application shown in Section 6 utilizes this property extensively.

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 any exist.
- 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 any exist. Default elements can be inserted only if declared as defaultable by the description.

A notion of *specificity* of defaults, reminiscent of CSS [4], 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. Instead, they are considered as applicable only to 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>
```

Analogously to CSS, 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.

Most other schema languages contain a default mechanism. However, some only support attribute defaults or content defaults that only contain chardata. Only DSD allows individual elements to be inserted in content sequences, and it is also unique in separating the default declarations from the structural descriptions. In Section 6, 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 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 unique mechanism allows a more precise description of semi-structured data. An example is the DSD notation itself, as shown in 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>
```

The ID definitions, IDRef references, and points-to requirements are checked in a separate phase after the main traversal of the application document.

3.8 Redefinitions and evolving DSDs

Many XML languages are built from existing languages. Also, often a whole family of related languages is to be defined. DSDs support these software practices by providing two simple mechanisms: *document inclusion* and *redefinition*. This allows schemas to be created from existing schemas through modifications and extensions.

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>
```

Most schema languages support modularization using `include`-like features. Some also allow redefinitions, but without being able to refer to the old definitions as our `CurrIDRef`. More details are given in Section 7.

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. Documentation may consist of arbitrary XML, but XHTML is recommended. This allows useful visual effects, such as showing the brief description in a box that pops up when the mouse is over the construct. Examples of documentation are shown in Section 6.

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 its syntax, 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. As noted in Section 5, the property of being entirely self-describable is not only esthetically pleasing, it is also

practically useful for application development. Furthermore, it supports development of schemas: the same tool that checks validity of application documents can be used to check that a given XML document is a valid DSD. Most other schema languages require separate tools for that, since they are not completely self-describable.

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 unique `title` to appear anywhere in the sequence of `book` elements. If we wanted to mandate the position of the `title` element, then a surrounding `Sequence` constructor was required. The remaining definitions are presented below, excluding the `title` element and the `isbn` string type that are shown in Section 3. We first show the definition of `book` elements:

```
  <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>
```

Note that the `isbn` attribute is optional. If it is not present in a `book`, then a `title` is mandatory. The definitions of `author` and `publisher` are as follows:

```
  <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. All `homepage` elements contains a URL:

```
  <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. The remaining string type definitions are as follows:

```
  <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 could be part of a standard library, constructed as a file containing numerous `StringTypeDef` elements that are accessed through the `include` mechanism. The following definition allows untitled books to receive the default title `Untitled`:

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

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 The DSD 1.0 Tool

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

Given the URI of an application document containing a DSD-reference processing instruction, the DSD tool performs the traversal of the application document as described in Section 3, and if it succeeds, it then performs the ID/IDRef and points-to checks 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.

By using a DSD processor as a front-end for other XML tools, these often become much simpler to construct since the subsequent phases may assume that the input satisfies the syntactic requirements defined by a certain DSD. This is exemplified by the IVR system described in the next section. The DSD processor itself relies on this technique. Using the meta-DSD, which is a complete description of the DSD language itself, the processor checks that a purported DSD document is indeed a DSD. This bootstrapping technique has reduced the size of the implementation considerably and made it more robust and readable.

The DSD processor analyzes application documents in linear time: execution time is proportional to the size of the application document. This assumes that the document size includes the inserted defaults, and that we view all default definitions, including those written in the application document, as belonging to the DSD. The constant of proportionality naturally depends on the complexity of the given DSD. This linear-time property makes the behavior of the DSD processor more predictable than with other schema language processors, where such guarantees are typically not required by the language specification.

6 Industrial Case Study: IVR Systems

IVR (Interactive Voice Response) systems range from simple telephony applications to complicated dialogue systems based on speech recognition. But even the simpler systems are notoriously difficult to construct since their programming involves complex timing and error issues. To simplify the task, many layers of abstractions are introduced. At the highest level, an application programmer chooses between ready-made dialogues, which are parameterized with prompts and timeout durations. In this section, we will show how XML and the DSD Schema may help an application programmer to learn and to use a specialized notation with many interdependent parameters such as prompts, timeout values, error counts, and error messages. In particular, we will show how a DSD processor automatically selects 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 [21]. The XPML notation has evolved from a simple variation of HTML, dubbed PML, to 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 in describing almost all syntactic constraints, while providing a practical solution to the handling of defaults. Indeed, questions of how to use PML effectively in practice 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.

6.1 The IVR scenario

Our case study is presented from the application programmer's point of view. The scenario calls for the development of a tiny interactive voice application that greets customers of different nationalities. The programmer will use the domain-specific language XPML, whose syntax is defined using DSD. The main idea of XPML 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 correspond 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.

6.2 DSDs for syntax explanations

Our application programmer is a novice XPML user, who has seen only a few examples of XPML code. One role of the DSD is to provide a readable, concise syntactic summary. The programmer should not have to read the DSD as an XML file; instead, a BNF-like version in the form of 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) is less appealing:

<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; it is rather complicated, taking up approximately 25 pages.

6.3 DSDs for debugging

We now explore how schemas may help the debugging of XML documents. Assume that the application programmer's first attempt at the XPML program is:

```
<?dsd URI="xpml-att.dsd"?>
<XPML>
  <head>
    <application name="HELLOWORLD"/>
    <maintainer address="klarlund@research.att.com"
                loglevel="2"/>
    <title>The Greeting Application</title>
  </head>
  <body>
    Welcome.
    <span nointerrupt="y">
      <audio url="/audioclips/greeting.vox"/>
    </span>
    <a name="repeat"/>
    <menu name="nationality">
      <option dtmf="0">To end</option>
      <do><a href="#endit"/>
         <comment>go to end point</comment>
      </do>
      <option> If you speak English. </option>
      <do> Hello!  How are you? </do>
      <option> If you speak Danish. </option>
      <do> Goddag!  Hvordan går det? </do>
    </menu>
    <a href="#repeat"/>
    <a name="endit"/>
  </body>
</XPML>
```

The programmer has inserted a `<?dsd URI="xpml-att.dsd"?>` processing instruction to indicate that the document must conform to the DSD named `xpml-att.dsd`. Using the DSD processor to check the syntax of the document will now produce the response:

```
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 the pertinent auxiliary definitions:

```
ConstraintDef ID=message-attributes:
  nointerrupt="YesOrNo"[Optional]

StringTypeDef ID=YesOrNo:
  ("yes" | "Yes" | "no" | "No")
```

revealing that the programmer must write `"yes"` instead of `"y"`. Naturally, the other schema notations offer similar capabilities. Most people will probably get acquainted with schemas only through such error-reporting; thus, it is very important that the schema notation itself is as simple as possible to make error messages understandable for non-experts.

6.4 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="klarlund@research.att.com"
                loglevel="2"/>
    <title>The Greeting Application</title>
  </head>
  <body>
    Welcome.
    <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 speak English. </option>
      <do> Hello!  How are you? </do>
      <option> If you speak Danish. </option>
      <do> Goddag!  Hvordan går det? </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 HTML layout, is dependent on a great number of parameters whose tuning is often essential to obtaining the desired performance. However, it would be quite a burden if all parameters should explicitly be stated in each document.

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 subsume CSS: in the domain of visual formatting there are some arithmetic rules about inheritance of values that cannot be expressed in DSD.

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 `explain`, the default content of `help` instructs the customer to press "1" to have the error explained.

```
<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="explain"/>
    </Element>
  </Context>
  <DefaultContent>
     <help>
       Press 1 for further explanation.
     </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 customized.

6.5 DSDs for simplifying XPML processing

With a DSD processor, 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.

Since defaults can only be overridden by application document, 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 permit it to 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.

6.6 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. This exercise has illustrated four practical aspects of DSD schemas:

- DSDs aid the XPML programmer to choose the right syntactic constructs. To enhance readability of DSDs, 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 that it may impede its use as an explanatory medium directed towards computer professionals.

7 Related Work

The specification of the basic XML notation includes the schema language DTD (Document Type Definition), which is a subset of the DTD mechanism known from SGML. XML DTD is a grammar notation that allows a content model and a list of attributes to be declared for each element name. The content model is a restricted form of regular expressions over element names and chardata nodes: if chardata is allowed, then only unordered content can be described. Also, content specifications have to satisfy a determinism property, which is reminiscent of our operational interpretation of content expressions. Attributes can be declared with another restricted form of regular expressions permitting only enumerations of strings to be specified. The attribute declarations also allow defaults to be specified. As in DSD, validity checking is performed by a top-down traversal of the application document. In addition to grammatical descriptions, DTD also contains the notions of *entity definitions*, which is a kind of macro mechanism, and *notations*, which provide semantic references to data formats. The typical use of entity definitions is subsumed by our inclusion mechanism and the various definition constructs. DSD, as well as many other schema languages, does not have an equivalent of DTD notations, since we regard them as independent of syntactical descriptions.

It is generally agreed that DTD is insufficient for many promises. Some typical arguments are: it does not itself use XML notation; most common data types for chardata and attribute values cannot be expressed; there is very limited support for modularity and reuse of descriptions; and content and attribute declarations cannot depend on attributes or element context.

A large number of other schema languages have been proposed since the introduction of XML and DTD. In the following, we give a brief summary of these, focusing on what we believe are the most important alternatives: XML Schema and RELAX NG. A further comparison of various schema language proposals, including DSD, can be found in [25].

7.1 XML Schema

Based on the experience with the DTD, XML-Data, DDML, DCD, and SOX schema languages, which we mention in Section 7.3, W3C has designed the language XML Schema. The requirements that this schema language should address according to W3C are found in [26]. This document briefly outlines usage scenarios such as publishing, electronic commerce transactions, authoring, databases, and metadata exchange, which are areas we believe are covered by DSDs. The design principles are summarized as follows: "The XML Schema language shall be more expressive than XML DTDs; expressed in XML; self-describing; usable by a wide variety of applications that employ XML; straightforwardly usable on the Internet; optimized for interoperability; simple enough to implement with modest design and run time resources; and coordinated with relevant W3C specs." Additionally, a number of structural requirements are defined: "The XML Schema language must define mechanisms for constraining document structure (namespaces, elements, attributes) and content (datatypes, entities, notations); mechanisms to enable inheritance for element, attribute, and datatype definitions; mechanism for URI reference to standard semantic understanding of a construct; mechanism for embedded documentation; mechanism for application-specific constraints and descriptions; mechanisms for addressing the evolution of schemata; and mechanisms to enable integration of structural schemas with primitive data types."

The DSD language, we believe, satisfies the principles and requirements outlined above, except that we have paid less attention to a precise coordination with other W3C standards (some of which were under development when DSD was designed). Laying aside issues such as whether XML Schema or DSD is straightforwardly usable on the Internet, we present next some significant technical and conceptual differences.

Constraints vs. complex types

The most essential difference between XML Schema or DSD is the way structural descriptions are specified. In DSD, the central notion is the *constraint*, which corresponds to the *complex types* in XML Schema. However, the constraints in DSD involve boolean logic and context expressions; neither feature has a counterpart in XML Schema.

The DSD constraint mechanism allows attribute declarations and content descriptions to depend on attributes in the the current element and of its ancestors. As shown in Section 3.2, this is a very useful mechanism that we believe many XML language descriptions can benefit from. In fact, both XHTML and the XML Schema language itself have validity requirements of this form, but they simply cannot be formalized in XML Schema.

Content models

The notion of complex types in XML Schema is also related to our content expressions. In XML Schema, the regular expression operators cannot be combined arbitrarily. For instance, the operator for describing unordered content can contain only individual elements. This makes it difficult to express combinations of ordered and unordered

content. Also, in XML Schema when describing mixed content, that is, content containing both elements and chardata, no constraints can be given on the chardata. Only if the content is pure chardata can its values be constrained. In DSD, these limitations do not exist.

String types vs. simple types

The string types in DSD correspond to the *simple types* in XML Schema defined in [2]. While we resort to regular expressions and user defined libraries of common type definitions, XML Schema is primarily based on a large number of predefined types and various derivation mechanisms. But, XML Schema UNIX-style regular expressions are also supported. The derivation operators can admittedly be more appropriate than standard regular expression operators, but the expressive power of these two approaches is formally the same. In our opinion, this sublanguage of XML Schema may be too complex relative to its benefits.

Inclusion and redefinition vs. inheritance and substitution groups

XML Schema and DSD also differ significantly in their approach to amending and reusing definitions. DSD uses a simple inclusion mechanism combined with selective redefinitions, while XML Schema contains a more complicated type system inspired by object-oriented programming languages. This type system contains two mechanisms: inheritance by extension or restriction along with substitution groups. The inheritance mechanism allows instance elements of a subtype in places where a supertype is required, but only if the elements are explicitly typed in the application document using special `xsi:type` attributes. Also, types can be defined to be abstract or final, as known from programming languages. The substitution group mechanism allows groups of types to be defined in a way that resembles the inheritance mechanism, but without the hierarchical type structure and explicit types. The DSD proposal does not rely on object-orientation, since we found that most application domains do not lend themselves to this paradigm and are better served with a simpler mechanism.

This raises the question of how DSDs may emulate inheritance. The answer is that a constraint describing the content of an element type may be extended to include more content according to an attribute describing the subtype. The constraint augmentation technique is much more flexible than derivation, for example, content may depend on more than one attribute. However, it does not offer the guarantee that a later definition will not violate the principle of object-orientation that an object of a subtype can be used wherever a supertype is expected.

Default insertion

The default mechanism in XML Schema is similar to the one of DTDs, except that default strings can be inserted in empty elements. In DSD, the defaults are not tied to the element and attribute declarations, but are instead defined by an association to a context expression that can query ancestors and attributes. If for a default definition this expression is true for the current element, the default is applicable. As shown

in Section 6.4, this mechanism may be quite useful in practice. Also, with DSDs default content is not inserted when an element is empty, but when a content expression requests it. In contrast to XML Schema, this mechanism allows defaults to be inserted in the middle of a content sequence, and it is not limited to chardata.

Self-describability

In general, a schema language is self-describing if and only if it is possible within that language to express all requirements for a document to define a valid schema. Such a self-description is called a *meta-schema*.

According to the design requirements, XML Schema was originally intended to be self-describable, however, the resulting language is not. As previously mentioned, it is not just a few technicalities that hinder this property: examples as the one in Section 3.2 can be found throughout the language specification. This precludes XML Schema from the many practical benefits of having a meta-schema similar to the one for DSD in Section 3.10. Additionally, it seems unsatisfactory to suggest an XML language intended to describe all common XML languages that cannot describe itself. In a sense, its complexity is higher than its expressiveness.

It is important to note that any schema language can be tweaked into being self-describable according to the above definition: instead of forbidding certain syntactic constructions, one can allow them all and just give them some obscure but well-defined semantics. The fact that DSD is self-describable does not imply that all valid DSDs are meaningful: since schemas capture requirements only about syntax or static semantics, there may always be semantic inconsistencies in a syntactically valid document. Still, in our experience the meta-DSD is able to catch most errors that occur while writing schemas.

Other features

XML Schema contains a few special features not mentioned so far. The notion of *nil values* in XML Schema allows elements to be empty despite of content requirements. Specifically, such an element must be declared nillable and have a `xsi:nil="true"` attribute. This feature can be emulated directly by our general conditional constraints.

XML Schema includes a subset of XPath [12] for expressing uniqueness requirements, keys, and references. A uniqueness constraint specifies that a given expression must be true at most once throughout a certain subset of the document. Keys and references are similar generalizations of the ID/IDREF concepts used in DTD and DSD. To keep the schema language simple, we have chosen not to include such general mechanisms in DSD.

7.2 RELAX NG

As a competitor to W3C's XML Schema, the RELAX NG language has emerged through a joining of the RELAX and TREX projects in an effort sponsored by OASIS. These languages all appeared after the DSD 1.0 specification was published.

RELAX [30] is based on the automata-theoretic characterization of regular tree languages formulated in [28]. According to the original RELAX concept, a specification expresses a nondeterministic bottom-up 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, even though they formally have less expressive power. There are several reasons for this decision. First, a top-down approach typically matches the hierarchical structure of the information being represented and thus is more natural to use. Second, bottom-up parsing prevents online processing, which requires a left-to-right traversal of the document text. Third, our idea that schemas should be a foundation for extending CSS to arbitrary XML requires that we use the same approach as CSS, which is top-down. Fourth, it is not obvious that the added expressive power is really necessary in practice. 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. Indeed, RELAX is suggested as a notation that is explicitly designed not to support default insertions. RELAX NG has inherited this lack of a default mechanism. But like the XML schema team, we believe that defaults must be supported by 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. However, our current semantics is formulated as a parsing process, not in terms of automata theory.

It was announced [29] that the RELAX project, influenced by the DSD notation, would adopt a top-down approach based on an automata-theoretic semantics. This has made the RELAX language quite similar to the TREX language [11], which has motivated the merge of the two projects.

A schema in RELAX NG is described by a top-down grammar, as in DSD. Elements are described by *patterns* corresponding to the notions of constraints in DSD and complex types in XML Schema. In contrast to XML Schema, RELAX NG contains a `choice` pattern operator, reminiscent of our boolean `Or` operator. However, neither the full boolean logic nor the conditional constraints are available, so complex dependencies may require all combinations of allowed attributes and contents to be spelled out. Also, RELAX NG does not contain a notion equivalent to our context expressions, so ancestor dependencies need to be encoded into the top-down grammar. This can cause a blow-up of the schema description when describing multiple ancestor dependencies. For instance, to simultaneously disallow nesting anchor elements and form elements, the grammar size will essentially increase by a factor of four.

RELAX NG relies on externally defined data types for attribute values and chardata. Only operators for enumerations and lists are built-in. In current implementations, the data types from XML Schema are supported, but this is not required by the specification. As in XML Schema, chardata cannot be constrained if describing mixed content—in contrast to DSD where this is possible.

We believe that RELAX NG has succeeded in providing a simple and expressive alternative to XML Schema. However, the lack of support for defaults, ancestor dependencies, and boolean logic may limit its usability.

7.3 Other proposals

DDML [6], which also has been called XSchema, was the result of a collaborative effort on the XML-DEV mailing list. It is a relatively straightforward generalization of DTD concepts using an XML notation, only adding little expressive power. A related language called DCD was proposed in [7]. It suggests using COBOL-like pictures for expressing string datatypes, adds cardinality constraints to the content models, and is formulated in the RDF framework. XML-Data [24] introduced element keys and references to generalize the ID/IDREF mechanism from DTD, and also inheritance of element descriptions for supporting modularization and reuse. Related to that is SOX [15], which is based more purely on an object-oriented paradigm. The XDR language [18] was designed as a simplification of XML-Data. None of these languages offer a unifying notion of context-dependent constraint as that in DSD.

Assertion Grammars [32] is an interesting approach that achieves some of our goals since it is based implicitly on nonterminals. It contains a powerful notion of tree patterns, which is reminiscent of our context patterns. 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.

The Schematron proposal [20] is based on idea of adapting the XPath framework for expressing tree patterns and validity constraints, as an alternative to grammar-centered formalisms. A Schematron schema consists of a number of declarative rules, each essentially being defined by two XPath expressions: a *context* specifying the applicability of the rule and a *assertion* specifying a validity requirement. Schematron language descriptions are open, in the sense that everything that is not explicitly forbidden is allowed. Most other schema languages, including DSD, have the opposite view. Because of the open description model and the high expressiveness of XPath, Schematron is often viewed as a supplement to ordinary schemas, not as a replacement.

As argued in Section 6.4, our form of default insertion is a useful way of assigning CSS-like properties in the form of element attributes to the application document. We know of no other work that has suggested a generalization of CSS based on a schema notation.

Finally, we compare DSDs to XSLT [10], which is a Turing-complete XML transformation language based on a tree-walking model. We have attempted to design DSD such that its expressive power matches essential aspects of this functional programming language. Specifically, it is very common that XSLT programs visit each node only once, recurse on children according to XPath tests that concern attributes of the current node and properties of ancestors, and carry no parameters. Generally speaking, DSDs can mimic the recursion of such XSLT programs. Technically, this can be proven by constructing a DSD constraint for each named or unnamed template. The function `xsl:apply-templates`, the basic recursive construct, can be translated into a constraint that drives typing of subnodes. The nodes selected are identified in the DSD through a propagation of types, where non-empty types are used in all

non-selected nodes. The details of this correspondence would require very technical arguments, which are outside the scope of this paper.

In XSLT, the programmer's assumptions about the existence or absence of attributes or children is implicit, and XSLT processors do not produce any error messages if such assumptions are not complied with. With DSD, the assumptions can be formalized and checked using the boolean logic and context-dependent constraint mechanisms.

8 Conclusion

The DSD language provides a simple but 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, declarative defaults, schema evolution, semi-structured data, complex data types, and efficient implementation. It has an expressive power that mirrors some essential aspects of XSLT. Moreover, the DSD language has been implemented and tested in practice. It is our hope that ideas from DSD may further simplify XML standards that go beyond just being grammar notations.

More concretely, we believe that in particular the following ideas have proven successful: the application of context expressions, boolean logic, and conditional constraints to describe dependencies, the flexible content model, the declarative default mechanism, and the top-down traversal method.

By the many proposals for XML schema languages, it is clear that there is no ideal solution to the problem of designing the right schema language that fits all purposes. We believe that the DSD language has succeeded in identifying the most central aspects of defining sets of XML document. Still, from the experience with DSD and other recent schema language proposals we are confident that the DSD language can be further simplified and yet become even more expressive in practice. Based on this, we continue the development of the DSD schema language in the future. As a first goal, DSD needs proper support for namespaces, as mentioned earlier.

Our implementation of a DSD processor is available in an open source package. 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 [22], example DSDs and application documents, and the XSLT style sheet for DSDs mentioned in Section 6.2.

Acknowledgments

We sincerely appreciate the extraordinarily thorough feedback that we received from the reviewers. We also thank the participants of the Spring 2000 XML course at the University of Aarhus for their enthusiasm. DSD users in the XML community have also provided us with many insightful comments and suggestions.

References

[1] Liora Alschuler. XML Schemas: Last word on last call, July 2000. http://www.xml.com/pub/a/2000/07/05/specs/lastword.html.

[2] Paul V. Biron and Ashok Malhotra, editors. *XML Schema Part 2: Datatypes*. W3C, May 2001. http://www.w3.org/TR/xmlschema-2/.

[3] Scott Boag et al., editors. *XQuery 1.0: An XML Query Language*. W3C, December 2001. http://www.w3.org/TR/xquery/.

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

[5] Ronald Bourret. Namespace myths exploded, March 2000. http://www.xml.com/pub/a/2000/03/08/namespaces/.

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

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

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

[9] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, editors. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C, October 2000. http://www.w3.org/TR/REC-xml.

[10] James Clark, editor. *XSL Transformations (XSLT) Specification*. W3C, 1999. http://www.w3.org/TR/WD-xslt.

[11] James Clark. TREX - tree regular expressions for XML, February 2001. http://www.thaiopensource.com/trex/spec.html.

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

[13] James Clark and Makoto Murata, editors. *RELAX NG Specification*. OASIS, December 2001. http://www.oasis-open.org/committees/relax-ng/.

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

[15] Andrew Davidson et al. *Schema for Object-Oriented XML 2.0*. W3C, July 1999. http://www.w3.org/TR/NOTE-SOX/.

[16] Steve DeRose, Eve Maler, and Ron Daniel Jr., editors. *XML Pointer Language*. W3C, September 2001. http://www.w3.org/TR/xptr.

[17] Steve DeRose, Eve Maler, and David Orchard, editors. *XML Linking Language*. W3C, June 2001. http://www.w3.org/TR/xlink.

[18] Charles Frankston and Henry S. Thompson. XML-Data reduced, July 1998. http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm.

[19] Philipp Hoschka, editor. *Synchronized Multimedia Integration Language (SMIL) 1.0 Specification*. W3C, June 1998. http://www.w3.org/TR/REC-smil.

[20] Rick Jelliffe. The Schematron: An XML structure validation language using patterns in trees, 1999. http://www.ascc.net/xml/resource/schematron/schematron.html.

[21] 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/software-symposium-ATT-00-paper/, September 2000.

[22] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. *Document Structure Description 1.0*. AT&T & BRICS, October 1999. BRICS NS-00-7, http://www.brics.dk/DSD/specification.html.

[23] 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.

[24] Andrew Layman et al., editors. *XML-Data*. W3C, January 1998. http://www.w3.org/TR/1998/NOTE-XML-data/.

[25] Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3), 2000.

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

[27] Anders Møller and Michael I. Schwartzbach. The XML revolution, December 2001. BRICS NS-01-8. http://www.brics.dk/~amoeller/XML/.

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

[29] Makoto Murata. Announcement on http://www.xmlhack.com, August 2000.

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

[31] Steven Pemberton et al., editors. *XHTML 1.0: The Extensible HyperText Markup Language*. W3C, January 2000. http://www.w3.org/TR/WD-html-in-xml.

[32] Dave Raggett. Assertion grammars. http://www.w3.org/People/Raggett/dtdgen/Docs/, May 1999.

[33] Jonathan Robie. W3C XML Schema questionnaire, July 2000. http://www.ibiblio.org/xql/tally.html.

[34] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, editors. *XML Schema Part 1: Structures*. W3C, May 2001. http://www.w3.org/TR/xmlschema-1/.