Document Structure Description 1.0*

Nils Klarlund† Anders Møller‡ Michael I. Schwartzbach†

Abstract

Document Structure Description 1.0 is a complete specification of a new XML notation for describing classes of XML documents. The notation is designed to be a simple tool based on familiar concepts. DSDs provide more flexible and precise structural descriptions than possible with DTDs or the current XML Schema proposal. A DSD generates a CSS-like default mechanism independent of formatting models. Finally, it allows an extension mechanism so that DSDs may be updated with new structural concepts.

1 Introduction

A *Document Structure Description (DSD)* is a specification of a class of XML documents. A DSD defines a grammar for XML documents, default element attributes and content, and documentation of the class. A DSD is itself an XML document. This note describes the design of the DSD notation.

1.1 Design goals

We have five major goals for the descriptive power of DSDs: they should

- allow context dependent descriptions of content and attributes, since the context of a node, such as ancestors and attribute values, often govern what is legal syntax;
- generalize CSS [4] (Cascading Style Sheets) so that readable, CSS-like rules for default attribute values and default content can be defined for arbitrary XML domains, not only predefined user formatting models;
- complement XSLT [9] in the sense that the expressive power of DSDs should be close to that of XSLT, so that assumptions made by XSLT style sheets can be made explicit in a DSD;

†AT&T Labs–Research, Shannon Labs, 180 Park Ave., Florham Park, NJ 07932. Email: `klarlund@research.att.com`.

‡BRICS, University of Aarhus, Ny Munkegade, DK-8000 Aarhus C, Denmark. Email: `{amoeller,mis}@brics.dk`.

- allow the description of semi-structured data, that is, the description of what references may point to; and
- allow the redefinitions of syntactic classes, so that language extensions can be expressed in terms of existing DSDs.

Currently, the W3C is preparing a similar notation called *XML Schema* [2]. In its current incarnation, XML Schema appears to address only the first goal and in a very limited way. DTDs (the grammar notation that is part of XML 1.0 [8]) meet none of these goals.

It is also important to us

- that a DSD yields a *linear time* algorithm for checking conformance of XML documents; and
- that DSDs are based on simple concepts that are familiar to computer scientists.

1.2 Design requirements

The essense of the DSD notation is captured through specific technical requirements that we believe are important to XML applications.

Syntax for attribute values Attribute values in XML documents are generally constrained, often in ways that cannot be expressed by enumeration types. Examples include data types like designations of time and font size, but also more sophisticated ones like lists of pairs of numbers, etc. Thus, the *syntax* (a.k.a. *data domains*) of attribute values must be expressible in a more general manner, namely with regular expressions. Regular expressions can be translated to deterministic finite-state automata; this representation is naturally efficient in most practical circumstances, at least for 7-bit alphabets. For example, dictionaries (large collection of words) become trees in the automaton representation. Work on the use of automata on large alphabets, such as in the MONA tool [12], indicates that the automaton representation may generalize well to Unicode.

Attribute dependencies Often attributes within an element are ordered in importance. In HTML, for example, the `maxlength` attribute of an `INPUT` element is relevant only when the `type` attribute has the value “text” or “password”. Thus, a concept of an *attribute dependency* must be explicitly provided. They are also essential to the insertion of defaults in the right order.

Boolean logic Frequently, as in the `maxlength` example, dependencies are boolean and involve the attribute values themselves. Therefore, *boolean logic* about attributes and their values must be part of the description of attribute dependencies. Also, conditionally required attributes should be expressible. For example, one might want to enforce the HTML 4.0 recommendation that for an `OBJECT` element a `type` attribute should be provided when the `data` attribute is used.

Content dependent on attribute values Content, the sequence of child elements of an element, sometimes depends on attributes or their values. For example, it may be required that the content of an element is empty when a `SRC` attribute is present. Such *content dependencies* must be expressible through content expressions that are linked to boolean logic.

Nonterminals DTDs and XML Schema suffer from the absence of nonterminals. If element descriptions are based on nonterminals, then multiple uses of the same element name can be described; also, the use of nonterminals is a natural glue for binding together several descriptions. For example, the mechanism for gluing modules together in the XHTML modularization note [1] is too simple: if a content model is to be extended, the original definition has to be literally included in the new one. In this way, it is impossible to extend a concept without knowing its full definition. Naturally, this problem severely restricts the composition of modules. Module composition based on the use of *nonterminals* solves this problem. Our use of nonterminals also allows DSDs to mimic the *processing modes* of XSLT [9].

Ordered content Ordered content must be described by regular expressions as in DTDs. However, since elements in the DSD framework are described by nonterminals, DSD content expressions are akin to extended BNF, that is, regular expressions over nonterminals.

Unordered content Element content often consists of several separate kinds of subcontent, where the ordering between kinds have no significance. For example, in the case of prompts in an XML-based dialog language, an element may require a help prompt and an error prompt, but their order is insignificant. Description of *unordered content* is required. Also, the mechanism must be efficient, that is, it must not entail product constructions of automata as in SGML.

Attribute defaults A simple CSS-like *default mechanism* must be provided that inserts attribute values according to attribute dependencies and context. It must be available in both DSDs and in application documents. It is not as simple as it appears to use a CSS-like formalism for XML documents in general. The problem is that CSS assumes a formatting model where *properties*, not attributes, carry the rendering details that are the outcome of running a CSS processor on a document. Thus, the outcome of a CSS style sheet applied to a document is not specified as an XML tree. The CSS model may entail that properties that are not appropriate for a particular node are in fact assigned, for example by the inheritance mechanism. (Certain properties are inherited, like `font-size`, even though that not all HTML elements make sense out of this property.) If attributes are identified with properties, then two problems must be addressed: (a) an order must be specified among attributes, as mentioned under "Attribute dependencies" above; (b) and attributes must be filled in only when they are actually allowed. Thus, in order to make a general CSS-like notation for XML, it is necessary to define it relative to an XML grammar description that determines an attribute order.

Element defaults Often, it is natural to specify element defaults. Example: in a markup language for voice dialogues, prompts are often left unspecified. They are best regarded as elements since they contain text augmented with text-to-speech markup. Thus, the default mechanism must also be able to insert *element defaults*. The element defaults may depend on attributes of the containing element, for example, if the available prompts depend on an attribute specifying the dialog structure. Also, as for attribute defaults, the element default insertion mechanism must be combined with the XML grammar.

Context dependencies Often, elements are allowed only in certain syntactic situations. For example, an `input` element in HTML with type "submit" is allowed only if it is contained in a `form` element. Such *context dependencies* must be expressible. The ability to express contexts should be similar to that of XSLT.

Extensibility of language description Frequently, platform variations or device differences are reflected at the level of the markup language. For example, some voice browsers may provide advanced controls of speech synthesis that are reflected in attributes like `SpeechRate`, but a language should not provide such attributes. Thus, a *language extension* mechanism must be available so that modifications to a language description can be precisely described.

Self-describability The notation must be *self-describable*: all static requirements to DSDs should be expressible in a DSD, called a *meta-DSD*.

Extension of DTD descriptive power The notation must *extend* the descriptive power of DTDs.

Documentability DSD *elements for documentation purposes* must be a part of the notation.

Easy collapse to elementary descriptions The addition of concepts, such as boolean logic, to the DSD notation when compared to other XML grammar proposals, must not complicate the presentation of a DSD to casual programmers. More concretely, it must be possible to create XSLT style sheets that extract the underlying context-free grammar behind a DSD while ignoring all boolean constraints.

1.3 Design non-requirements

We will omit certain aspects of XML DTDs. We find, as does the XML Schema Working group, that XML *notations* and *unparsed entities* are baroque. We draw the conclusion that they should be left out. We also leave out *parsed entities*, better known as parameterless macros or text macros. *Internal parsed entities* (macro definitions in the application document) may still appear in a document type declaration, which is allowed by virtue of the application document being XML. Similarly, *external parsed entities* (macro definitions in the XML Schema or DTD) will be allowed also only through the fact that the document is an XML document. The *parameter entities* of DTDs (yet another idiosyncratic text macro concept) should be made unnecessary by our use of nonterminals. Similarly, *conditional sections* should be made unnecessary by our extensibility requirement. Also, we will omit the IDREFS type, which could be easily reintroduced. We will omit tokenized (NMTOKEN and NMTOKENS) attribute types and enumerated attribute types; both are representable using regular expressions.

We would like to avoid using the elaborate vocabulary outlined in the XML Information Set specification draft [11]. We think that a grammar notation should be explainable in elementary terms.

We are not concerned with verbosity of DSDs; at least not to the extent it's caused by the inherent volume of parenthesized notations in XML.

1.4 Brief comparison to other proposals

DCDs [6] are more expressive than DTDs; so are DDMLs [5] and XML Schemas [8]. Neither of these extensions satisfy the majority of our goals. For example, they fail to specify attribute structure, relationships between attribute values and content, and content that depend on context. XML Schema, in the data description part [3], provides a variety of mechanisms for attribute value domains. In the structure description part [2], XML Schema provides mechanisms that in some regards are similar to ours; for example, *archetypes* are somewhat similar to our constraints.

The *assertion grammars* outlined in [13] elevate certain DTD parameter entities to a first-order status that allow content models to be extended piecemeal; in essence, this mechanism can be seen as a specialized way of redefining content expressions in DSDs. Assertion grammars, like DSDs, offer context sensitive ways of specifying content and attributes. They also provide an inheritance mechanism, which we believe could be adapted to our framework.

1.5 The future

In Section 1.3, we already mentioned a couple of issues that are currently left out and that we are working on solving.

XML namespaces [7] will not be addressed here, but we are working on a solution along the lines of [2].

We are also considering whether DSDs should be allowed to determine that certain attributes are inherited (in the sense of an inherited property of CSS). In addition, we are awaiting the final XSLT 1.0 specification, since it may be desirable to include the node set expression language, XPath, in DSDs.

1.6 The rest of the note

This note presents the complete design of the DSD language. Section 1.7 summarizes XML concepts and Section 1.8 provides an overview of the DSD concepts. The DSD language is described in detail in Section 2. Appendix A provides a meta-DSD, that is, a DSD for DSD documents. Appendix B shows an alternative concrete syntax for our CSS-like notation for context descriptions.

1.7 XML concepts

The reader is assumed to be familiar with XML [8]. An XML document (assumed to be well-formed) may be represented as a tree, called the *XML tree*. The XML tree's *internal nodes* are those that have *child nodes*; they correspond to *non-empty elements*. Child nodes are ordered. The *root* of the tree corresponds to the root element. *Leaf nodes* are those without children. They correspond to *empty elements*, *chardata*, *comments*, and *processing instructions*. *DTD information* is not represented.

Each element has a *name* and contains a set of *attributes*, which are pairs of *names* and *values*. Attributes are not regarded as nodes.

A *path* is a sequence, possibly empty, of nodes $v_1, \ldots, v_n$ such that v_i is a parent of v_{i+1}. We say that an element v occurs *before* element v' if the start tag of v occurs before the start tag of v' in the XML document. In particular, any element occurs before any of its descendants. The *last* element of some set of elements is defined using the same ordering of nodes.

The *content* of an element is the sequence of its child nodes. The *context* of a node is the path of nodes starting at the root of the document tree to the node itself. Attribute values and chardata nodes are strings of characters.

Processing instructions with target `dsd` or `include` are assumed to contain information relevant to the DSD processing of the document. All other processing instructions, comments, and chardata nodes consisting of white-space only, are ignored during processing but kept in the document.

Elements and attributes whose name has namespace prefix [7] `DSD` and namespace name `http://www.brics.dk/DSD` as well as declarations of this namespace occuring in the application document are removed before the actual processing. Elements named `DSD:Default`, however, are considered a part of the DSD and are recorded before being removed.

We often abuse language by not distinguishing between an XML concept and its tree representation; for example, an "element" often means a node representing the element. To avoid confusing the DSD and the application document, we usually refer to attributes of DSD elements as *properties*.

See [8] for a further explanation of XML concepts.

1.8 DSD concepts and overview

This section describes all the main concepts of the DSD notation and how they fit together. Several important details, described in Section 2, are omitted here.

A DSD is *valid* if all validity requirements in Section 2 hold. All DSDs are assumed valid, unless otherwise indicated. An *application document* is an XML document that is intended to be *conforming* to a DSD (as defined below). *Application document processing* is the process, carried out by a *DSD processor*, of checking the conformance of an application document to a DSD. The meta-DSD in Appendix A is constructed so that a DSD is valid if and only if it conforms to the meta-DSD.

Element IDs and descriptions A DSD defines a set of *element IDs*, which play a role similar to nonterminals in the parsing of context-free languages. During application document processing, each element in the document is assigned an element ID. The element IDs are assigned in a top-down manner starting with the root element ID determined by the DSD. Elements with the same name may be assigned different IDs during this processing. Each element ID determines an *element description*, which is a pair consisting of an element name and a constraint.

Note that many nodes in the application document may be assigned the same element ID. Thus, assigned element IDs do not identify individual nodes in the application document; they only do so in the DSD.

Constraints A *constraint* defines the attribute structure of the element, including the syntax of attribute values and dependencies among attributes and their values; a constraint also determines the structure of the content, which may depend on attribute values and context. During document processing, several constraints may be evaluated for a given element. During evaluation of a constraint, attributes and content are gradually declared.

Boolean logic and context patterns *Boolean expressions* describe properties about the context, the presence or absence of declared attributes, and their values. Context properties are described by *context patterns*, which are a kind of regular expressions over element names. Boolean expressions are *true* or *false*; they have no side effects.

Content expressions The content of an element is described by regular expressions, called *content expressions*, over element IDs. A string derived from the content expression determines, through the pairs determined by element IDs, a string of element names, where each name occurrence is associated with a constraint. The DSD language prescribes an operational way of deriving such a string for the content of an element.

String types String types are regular expressions over characters. They are used to characterize both attribute values and chardata nodes.

Evaluation An application document is processed by *evaluating* the element description denoted by the root element ID on the XML tree in what is essentially a single top-down pass. As a side-effect, the evaluation transforms the tree by inserting new attributes and nodes where required by defaults. For each element, the assigned element description is *evaluated*. The evaluation either *succeeds*, and the element *satisfies* the element description, or it *fails*. The evaluation of an element description on an element, called the *current element*, proceeds as follows:

1. It is verified that the name of the element description is the same as that of the current element; otherwise, the evaluation fails.

2. The constraint of the element description is evaluated as detailed in Section 2.7. During this evaluation, child elements are assigned element IDs according to the evaluation of the content expressions that are enforced by the constraint.

3. The child elements are evaluated recursively.

If any of these evaluations fail, the evaluation of the current element fails; otherwise, it succeeds.

Default insertions Default element attributes and content are associated a boolean expression. Defaults are only applicable when their boolean expression is true.

The insertion of default attributes depends on the structure of attribute declarations: the evaluation of the current element gradually declares attributes and only defaults that mention declared attributes are considered. In this way, priorities among attributes of an element can be expressed, and it is guaranteed that values are inserted only for allowed attributes.

Defaults can be specified both in the DSD document and in the application document. A notion of specificity takes care of assigning priorities to defaults whenever more than one is applicable. For instance, application document defaults always have higher priority than DSD document defaults.

ID types A DSD may declare that application document attributes are of type `ID` or `IDRef`. These characterizations are equivalent to DTD types ID and IDREF. Thus, an element possessing an attribute of type `ID` with value *id* is set to be a definition of *id*. A DSD may allow an attribute to redefine *id* through an element containing an attribute of type `RenewID` with value *id*. A reference of type `IDRef` with value *id* denotes the last definition in the document with value *id*. Similarly, a reference of type `CurrID` refers to the current definition, that is, the last definition occurring before the reference. The DSD may impose a boolean expression, called the *points-to requirement*, on the element denoted by a reference.

DSD concepts and definitions All main concepts of DSDs—except element descriptions—may be separately defined and redefined. The internal reference mechanism of DSDs themselves is based on the four ID types.

Document inclusion A simple document inclusion mechanism allows both DSDs and application documents to be created as extensions of other documents. During parsing, include declarations are replaced by the documents they refer to.

Conformance The application document is *conforming* if the evaluation of the root element of the application document succeeds, and each attribute that is a reference denotes a definition that satisfies the points-to requirement of the attribute. For a conforming document, the *result* of the application document processing is an XML document that is a textual representation of the transformed tree. The result may also include various information calculated during the processing.

2 The DSD language

This section describes the DSD language. We believe that all design goals and requirements mentioned in Sections 1.1 and 1.2 are fulfilled by this language.

The structure of a DSD is defined in this section by a traditional context-free grammar, since it is easier to read than a DTD. We use a form of Extended Backus-Naur Form (EBNF) notation in this note. (It is essentially the same as the one defined in Section 6 of the XML 1.0 specification [8].) For the sake of clarity, the syntax for empty elements is shown in the single tag form; also, attributes do not need to it appear in the order indicated, and single quotes may be used instead of double quotes. A DSD is a string that is derivable from *maindsd* according to the syntax in Sections 2.4 to 2.13 and that is a well-formed XML document.

2.1 DSD references

An application document refers to a *main DSD* by a processing instruction of the form

`<?dsd URI=`*AttValue*`?>`

appearing in the prolog (as defined in Section 2.8 of the XML 1.0 specification [8]) of the document. (Processing instructions with target `dsd` occuring after the prolog are ignored.) The reference indicates that the application document is intended to conform to the main DSD specified by the URI *AttValue*.

2.2 Document inclusions

Include processing instructions allow both DSD documents and application documents to be created as extensions of other documents.

`<?include URI=`*AttValue*`?>`

The `URI` property contains the URI of the included document. An *include replacement* consists of replacing the `include` processing instruction with the root element of the XML document designated by the URI. Any document can be, directly or indirectly, included only once into a given document. Include instructions with a URI that has already been included are replaced by the empty string. Document inclusion takes place before the actual processing in the order of occurence.

2.3 Internal definitions and references

Internal definitions and references are those that appear in the DSD; they are so called to distinguish them from definitions and references in the application document. When confusion is not possible, we omit "internal" when referring to them.

When a DSD element is of the form `<`*`S`*`Def ID="`*`id`*`"/>`, where *`S`* is `Element`, `Constraint`, `AttributeDecl`, `Content`, `Bool`, `Context`, or `StringType`, we say that it is an *internal definition* of *`id`*. Similarly, a DSD element of the form `<`*`S`*`Def RenewID="`*`id`*`"/>` is called an *internal redefinition*.

A DSD element of the form <*S* IDRef="*id*"/> is an *internal, final reference* to *id*. Similarly, a DSD element of the form <*S* CurrIDRef="*id*"/> is called an *internal, current reference*.

The *target element* of a final reference to some *id* is the last definition or redefinition of *id*. The target element of a current reference to some *id* is the last definition or redefinition of *id* that occurs before, but does not contain, the internal current reference. An internal redefinition is both considered a definition and a reference. The target of an internal renewing definition is defined as for an internal current reference.

Validity requirements For every reference <*S* IDRef="*id*"/> or <*S* CurrIDRef= "*id*"/>, the target must exist and be a structural definition of the appropriate form, that is either of the form <*S*Def ID="*id*"...>...</*S*Def> or of the form <*S*Def RenewID= "*id*"...>...</*S*Def>. There may be at most one definition of a given *id* in a DSD. Before each redefinition, there must occur a definition of the same *id* that does not contain the redefinition.

2.3.1 The meaning of internal references

During processing, an *ID stack* (which is initially empty) is used to keep track of the use of internal final references. An internal final reference to some *id* is *processed* as follows:

- If the *id* does not appear in the ID stack,
    1. the *id* is pushed onto the ID stack,
    2. the target is processed in place of the reference, and
    3. the *id* is popped off the ID stack.
- Otherwise, if the *id* already appears in the stack, the meaning of the reference depends on its form as follows. For a `Bool`, the reference evaluates to false; for a `Context`, the match fails; for a `StringType`, the denoted language is the empty one; for a `Constraint`, evaluation succeeds; and for a `Content`, the tentative evaluation succeeds and no content is consumed.

This mechanism avoids self-referencing definitions. Internal current references are processed without using the stack, just by processing the target of the reference.

2.4 Main DSD

A DSD consists of various meta-information, subDSDs, defaults, and structural definitions. SubDSDs typically arise from the use of `include` processing instructions.

maindsd → `<DSD IDRef=`*AttValue* `DSDVersion=`*AttValue*`>`
    dsdcontent
  `</DSD>`

subdsd → `<DSD` (`IDRef=`*AttValue*)? `DSDVersion=`*AttValue*`>`
    dsdcontent
  `</DSD>`

dsdcontent → (`<Title>` *content* `</Title>`)?
  (`<Version>` *content* `</Version>`)?
  (`<Author>` *content* `</Author>`)?
  (*doc* (*subdsd* `|` *default* `|` *structdef*))∗

The `IDRef` property of the *maindsd*, called the *root element ID*, denotes the *root element description* that an application document must satisfy. The `DSDVersion` must be specified. To comply to the present specification, it must have the value "`1.0`". The elements `Title`, `Version`, and `Author` may be used to specify meta-information about the DSD; they may contain arbitrary well-formed XML, but a simple HTML subset like [10] is recommended.

Example

Here is a complete DSD that requires an XML document to contain a single element `Hello` whose content is character data:

```
<!-- http://www.brics.dk/DSD/examples/1.0/hello_world.dsd -->
<DSD IDRef="Hello" DSDVersion="1.0">

  <Title>HelloML</Title>

  <ElementDef ID="Hello">
    <StringType/>
  </ElementDef>

  <Default>
    <Context><Element Name="Hello"/></Context>
    <DefaultContent>
      Hello world!
    </DefaultContent>
  </Default>

</DSD>
```

The DSD also defines the default value of character data inside `Hello` elements to be the string "Hello world!". So, the application document

```
<?dsd URI="http://www.brics.dk/DSD/examples/1.0/hello_world.dsd"?>

<Hello/>
```

conforms to the DSD, and the document resulting from the DSD processing is

```
<?dsd URI="http://www.brics.dk/DSD/examples/1.0/hello_world.dsd"?>

<Hello>
  Hello world!
</Hello>
```

2.4.1 Structure definitions

A *structdef* binds a structure description to an ID.

structdef → *elementdef* |
constraintdef |
attributedecldef |
contentdef |
booldef |
contextdef |
stringtypedef

2.4.2 Documentation

Documentation may be associated to all definitions and to most other significant syntactic constructs. Documentation elements do not affect the processing.

doc → (`<Label>` *content* `</Label>`)?
(`<BriefDoc>` *content* `</BriefDoc>`)?
(`<Doc>` *content* `</Doc>`)?

The form `Label` can be used to assign a label to the syntactic construct. The `BriefDoc` may be used for brief descriptions (such as those used to display a message when the mouse cursor is placed over a hyperlink). The `Doc` can be used for a longer explanation (such as for an interactive manual). A documentation element may contain arbitrary well-formed XML elements and chardata, but using XMLized HTML is recommended.

2.5 Defaults

A *default* associates a set of *default attributes* and *elements* to a boolean expression.

default → `<Default>`
(*doc* *boolexp*)?
(*defattribute* | *defcontent*)∗
`</Default>`
defattribute → `<DefaultAttribute Name=`*AttValue* `Value=`*AttValue*`/>`
defcontent → `<DefaultContent>`
(*element* | *CharData*)
`</DefaultContent>`

A `DefaultAttribute` defines a default value for attributes of the given name. A `DefaultContent` defines a default element or chardata node. If it contains an element, this element is a default for elements with the same name. *CharData* is regarded as a default for the *chardata* pseudo-element, see Section 2.10.

A default is *applicable* for an attribute name (or element name) if

- it contains a default attribute (or element) with the same name; and
- either its boolean expression is omitted or it is meaningful and true.

When more than one default is applicable, the *selected default* for an attribute name (or element name) is found by narrowing the set of applicable defaults to the ones with the highest specificity among them (see Section 2.5.2). Then, the latest occurring default definition is the selected default if the set is non-empty; otherwise, the selected default does not exist.

Default attributes and content are inserted according to Section 2.8 and Section 2.10, respectively.

2.5.1 Application document defaults

Defaults can also be specified in the application document. Every application document element may contain `DSD:Default` elements that extend the DSD.

appdocdefault → `<DSD:Default>`
  (*doc* *boolexp*)?
  (*defattribute* `|` *defcontent*)∗
`</DSD:Default>`

The namespace `DSD` must be declared with the name `http://www.brics.dk/DSD`. According to the explanation below, a default defined in the application document is applicable only to the parent of the `DSD:Default` node and to the content of the parent. Internal current references occuring within a `DSD:Default` are treated as internal final references.

2.5.2 Specificity of defaults

The specificity of defaults is defined by the following rules:

- A default defined in the application document has higher specificity than any default defined in the DSD document.
- The specificity of defaults defined in the DSD document is defined by the specificity of their boolean expressions (see Section 2.11.1). If the boolean expression is omitted, then the default implicitly has the minimum specificity.
- For application document defaults, a default d_1 has higher specificity than a default d_2 if the parent of the `DSD:Default` element containing d_1 is a descendant of the parent of the `DSD:Default` element containing d_2; in the case that they have the same parent, their specificity is determined according to the rules for defaults defined in the DSD document.

Example

Speech applications often require tuning of parameters that depend—at the same time—on the platform (such as the speech engine) and on programmer-defined abstractions (such as class attributes). Here, we define a DSD fragment that can be included as a stylesheet in speech documents:

```
<DSD:Default>
  <Context><Element Name="select"/></Context>
  <DefaultAttribute Name="ATT:speech-engine" Value="watson.2039"/>
</DSD:Default>

<DSD:Default>
  <Context>
    <Element Name="select">
      <Attribute Name="interaction_class" Value="myown"/>
      <Attribute Name="ATT:speech-engine" Value="watson.2039"/>
    </Element>
  </Context>
  <DefaultAttribute Name="ATT:speech-engine-timeout" Value="2s"/>
</DSD:Default>
```

This stylesheet instructs the DSD processor to augment `select` elements with an `ATT:speech-engine` attribute designating the `watson.2039` speech engine. Moreover, it tells the DSD processor to augment all `select` elements whose `interaction_class` attribute is `myown` and whose `ATT:speech-engine` attribute is `watson.2039` with an attribute `ATT:speech-engine-timeout` whose value is `2s`.

Note that a programmer would expect the `ATT:speech-engine` attribute to be inserted before the `ATT:speech-engine-timeout` attribute; see the example in Section 2.8 for how such priorities are specified.

The DSD default syntax is nothing but an XMLized adaptation of Cascading Style Sheets. A more familiar concrete syntax would be:

```
select {'ATT:speech-engine': watson.2039}

select [interaction_class='myown']
       [ATT:speech-engine='watson.2039']  {'ATT:speech-engine-timeout': 2s}
```

2.6 Element descriptions

An *element description* defines a pair consisting of an element name and a constraint. An *element definition* associates an element description to an element ID.

elementdescr → `<Element IDRef=`*AttValue*`/>` |
`<Element Name=`*AttValue* (`Defaultable="`*YesOrNo*`"`)?`>`
  doc constraintexp
`</Element>`

elementdef → `<ElementDef ID=`*AttValue* (`Name=`*AttValue*)?
    (`Defaultable="`*YesOrNo*`"`)?`>`
  doc constraintexp
`</ElementDef>`

An element description of the first form (with an `IDRef`) is called *indirect* and of the second form (with a `Name`) *direct*. Every direct element description is assigned an implicit element ID, different from all other IDs, such that all element descriptions have an ID. The `Name` property defines the *name* of the description. If omitted, the name defined is the value of the `ID`.

An element description is evaluated on the current element as follows:

1. If the name of the current element is not the same as the name defined by the element description, then evaluation fails.

2. The constraint of the element description is evaluated according to Section 2.7. During this evaluation, attributes and content expressions are gradually declared. Initially, the sets of declared attributes and content expressions are empty. The element description evaluation fails if the evaluation of the constraint fails.

3. An evaluation error occurs if not all of the attributes of the current element are declared during the evaluation.

4. Similarly, an evaluation error occurs unless the sets of element names of declared content expressions are disjoint and their union contain the names of the element's child elements. This requirement treats character data as a pseudo-element. In particular, at most one of the content expressions may declare character data. Note that omitting content expressions from an element description has the effect that no content is allowed.

5. By the preceding requirement and the semantics of content descriptions (Section 2.10), each child element, except those matched by `AnyElement`, has been assigned an element ID. For each such child element, the element description designated by the ID is evaluated. If any of these evaluations fail, then the evaluation of the current element fails.

The meaning of the `Defaultable` property is defined in Section 2.10.

Example

We develop here a DSD for a markup language that expresses simple menus as found in IVR systems (IVR means Interactive Voice Response). The notation is based on a `select` element similar to that of HTML:

```
<!-- http://www.brics.dk/DSD/examples/1.0/phone_example.dsd -->
<DSD IDRef="phoneml" DSDVersion="1.0">

  <Title>PhoneML DSD</Title>

  <ElementDef ID="phoneml">
    <OneOrMore>
      <Element IDRef="select"/>
    </OneOrMore>
  </ElementDef>

  ...
</DSD>
```

This DSD, which is not yet complete, defines the markup language to consist of a root element identified by the element definition `phoneml`. This element description defines the element name to be `phoneml` (since no explicit name is provided in the element definition). It also defines the content to be a non-empty sequence of elements provided by the `select` element definition, which is:

```
<ElementDef ID="select">
  <AttributeDecl Name="class" Optional="yes"/>
  <OneOrMore>
    <Element IDRef="option_element"/>
  </OneOrMore>
  <Element Name="help" Defaultable="yes">
    <Content IDRef="audio_content"/>
  </Element>
  <Element Name="prompt" Defaultable="yes">
    <Content IDRef="audio_content"/>
  </Element>
</ElementDef>
```

The constraint of this definition declares an optional `class` attribute; it also introduces three kinds of content, which can be arbitrarily interspersed: one or more elements defined by `option_element`, a `help` element with content `audio_content`, and a `prompt` element, also with content `audio_content`. The `option_element` element definition is:

```
<ElementDef ID="option_element" Name="option">
  <AttributeDecl Name="value"/>
  <Content IDRef="audio_content"/>
</ElementDef>
```

which defines elements with ID `option_element` to have the tag name `option` and content `audio_content`. So, if we assume that the elements `option`, `help`, and `prompt` may contain character data, then a conforming document might be

```
<?dsd URI="http://www.brics.dk/DSD/examples/1.0/phoneml.dsd"?>
<phoneml>

  <select>
    <option value="blue">blue widget</option>
    <option value="red">red widget</option>
    <prompt>Please select red or blue widget!</prompt>
    <help>Say red or blue</help>
  </select>

</phoneml>
```

The order of appearance of the `help` and `prompt` elements within the `select` element is inconsequential. Here is another conforming way of writing the `select` element:

```
<select>
  <help>Say red or blue</help>
  <option value="blue">blue widget</option>
  <prompt>Please select red or blue widget!</prompt>
  <option value="red">red widget</option>
</select>
```

2.7 Constraints

A *constraint definition* associates an ID to a constraint expression. A *constraint expression* is a sequence of *constraint terms*.

constraint → `<Constraint` (`IDRef=`*AttValue* `|` `CurrIDRef=`*AttValue*)`/>` `|`
`<Constraint>` *doc constraintexp* `</Constraint>`

```
constraintdef  → <ConstraintDef (ID=AttValue | RenewID=AttValue)>
                   doc constraintexp
                 </ConstraintDef>
constraintexp  → (doc constraintterm)*
constraintterm → attributedecl |
                 condconstraint |
                 contentexp |
                 boolexp |
                 constraint
condconstraint → <If> doc boolexp
                   <Then> doc constraintexp </Then>
                   (<Else> doc constraintexp </Else>)?
                 </If>
```

A constraint that is a reference is evaluated according to Section 2.3. Evaluation of a constraint that is not a reference consists of evaluating its constraint expression. The evaluation of a constraint expression consists of evaluating its constraint terms in their order of occurrence. If any such evaluation fails, then the evaluation of the constraint fails.

Constraint terms are evaluated as follows:

- Attribute declarations are evaluated as described in Section 2.8.
- For a conditional constraint, its first constraint expression is evaluated if the boolean expression is true; otherwise, and if it contains a second expression, this expression is evaluated.
- Content expressions are evaluated as described in Section 2.10.
- A boolean expressions is verified to be meaningful and true; otherwise, the evaluation fails.
- A *constraintterm* being a *constraint* is evaluated as the constraint expression it contains.

Note that both a *condconstraint* and a *contentexp* may take the shape of an `If` element. The ambiguity is solved by always regarding an `If` element in this context as a *condconstraint*.

Example

We describe how to make the PhoneML language of Section 2.6 extensible so that we can later add new attributes to the `select` element. Since element IDs cannot be redefined, we must in the initial design of the DSD make a "hook" for extensions. The hook is a constraint that we name `select_constraint`. We change the DSD as indicated here:

```
<!-- http://www.brics.dk/DSD/examples/1.0/phone_with_constraints.dsd -->
<DSD IDRef="phoneml" DSDVersion="1.0">

  <Title>PhoneML DSD</Title>
```

```
  ...
  <ElementDef ID="select">
    <Constraint IDRef="select_constraint"/>
  </ElementDef>

  <ConstraintDef ID="select_constraint">
    <AttributeDecl Name="class" Optional="yes"/>
    <OneOrMore>
      <Element IDRef="option_element"/>
    </OneOrMore>
    <Element Name="help" Defaultable="yes">
      <Content IDRef="audio_content"/>
    </Element>
    <Element Name="prompt" Defaultable="yes">
      <Content IDRef="audio_content"/>
    </Element>
  </ConstraintDef>
  ...
</DSD>
```

2.8 Attribute declarations

An *attribute declaration* declares an attribute with a name and a type.

attributedecl → <AttributeDecl (IDRef=*AttValue* | CurrIDRef=*AttValue*)/> |
<AttributeDecl Name=*AttValue* *attrdeclattrs* >
attrdeclcontent
</AttributeDecl>

attributedecldef → <AttributeDeclDef (ID=*AttValue* | RenewID=*AttValue*)
Name=*AttValue* *attrdeclattrs* >
attrdeclcontent
</AttributeDeclDef>

attrdeclattrs → (Optional="*YesOrNo*")? (IDType="*IDType*")?

attrdeclcontent → (*doc* *stringtypeexp*)? (*doc* *pointsto*)?

IDType → ID | IDRef | RenewID | CurrIDRef

pointsto → <PointsTo> *doc* *boolexp* </PointsTo>

An attribute declaration that is a reference is evaluated according to Section 2.3. If it is not a reference, then it is evaluated as follows on the current element. If its name is already in the current set of declared attributes, then the evaluation fails. Otherwise, if the attribute (as named in the declaration) is not present in the current element, and there are applicable defaults then the selected default is inserted. The evaluation is successful if either,

- the attribute is not present (after possible default insertion) and the declaration has the Optional property with value "yes" or "Yes", or
- the attribute is present and either the declaration does not provide a string type or the attribute value satisfies the string type according to Section 2.13.

Otherwise, if the attribute is present and does not satisfy the string type, evaluation fails. If evaluation is successful, then the attribute is said to have been *declared*.

If an `IDType` has been specified there are some additional requirements in order for evaluation to be successful:

- no two attributes of type `ID` in the document can have the same value;
- the value of an attribute of type `IDRef` must occur as the value of an attribute of type `ID` of some other element; and
- the value of an attribute of type `RenewID` or `CurrIDRef` must occur as the value of an attribute of type `ID` of some element that is before and that does not contain the current element.

If evaluation is successful, an attribute of type `IDRef` is a *reference* to the last element that has an attribute of type `ID` or `RenewID` with the same value. An attribute of type `RenewID` or `CurrIDRef` is a reference to the last element that is before and does not contain the current element and has an attribute of type `ID` or `RenewID` with the same value.

The semantics of `PointsTo` is explained in Section 2.15.

Validity requirements A `PointsTo` may be present only if the attribute declaration has the `IDRef`, `RenewID`, or `CurrIDRef` type. An attribute declaration may have the `Optional` property only if it is not an `IDType`.

Example

The attribute declaration below declares an attribute with name `publication_ref`. Its value is an ID reference, whose referenced element must have the name `Book` or `Article`.

```
<AttributeDecl Name="publication_ref" IDType="IDRef">
  <Stringtype IDRef="Name"/>
  <PointsTo>
    <Or>
      <Context><Element Name="Book"/></Context>
      <Context><Element Name="Article"/></Context>
    </Or>
  </PointsTo>
</AttributeDecl>
```

Example

Here we use the PhoneML description of Section 2.7 to add platform and programmer specific attributes. Insertion of defaults for these attributes were discussed in Section 2.5.

```
<!-- http://www.brics.dk/DSD/examples/1.0/phone_extension.dsd -->
<DSD IDRef="phoneml" DSDVersion="1.0">

  <Title>PhoneML DSD (with AT&amp;T engine and "interaction_class")</Title>

  <?include URI=
     "http://www.brics.dk/DSD/examples/1.0/phone_with_constraints.dsd"?>

  <ConstraintDef RenewID="select_constraint">
    <Constraint CurrIDRef="select_constraint"/>
```

```
    <AttributeDecl Name="interaction_class" Optional="yes"/>
    <AttributeDecl Name="ATT:speech-engine" Optional="yes"/>
    <AttributeDecl Name="ATT:speech-engine-timeout" Optional="yes"/>
  </ContentDef>

  <Default>
    <Context><Element Name="select"/></Context>
    <DefaultAttribute Name="interaction_class" Value="myown"/>
  </Default>

</DSD>
```

Note that the `interaction_class` attribute is introduced before the `ATT:speech-engine` attribute, which in turn is before the `ATT:speech-engine-timeout` attribute. This order in the DSD also specifies the order in which defaults are inserted; refer to the example in Section 2.5 to see why this is important.

2.9 Attribute descriptions

An *attribute description* consists of a name and a string type.

```
attributedescr → <Attribute Name=AttValue Value=AttValue/> |
                 <Attribute Name=AttValue>
                   (doc stringtypeexp)?
                 </Attribute>
```

An attribute description is *meaningful* only for a declared presence of an attribute. The description is true if an attribute with the name denoted by `Name` is present and the attribute value either is the same as the name denoted by `Value` or satisfies the *stringtypeexp*. If neither a `Value` nor a *stringtypeexp* is specified, then every attribute value satisfies the description.

Example

Attribute descriptions may be used to guide the insertion of defaults, just as in CSS, see earlier example in Section 2.5. They are also useful in common situations where certain attribute combinations are disallowed. For example, we may express the requirement that exactly one out of the two attributes `ID` or `RenewID` must be present as the constraint:

```
<OneOf><Attribute Name="ID"/><Attribute Name="RenewID"/></OneOf>
```

2.10 Content descriptions

A *content description* defines a set of content by a *content expression*.

```
contentdescr → <Content (IDRef=AttValue | CurrIDRef=AttValue)/> |
               <Content> doc contentexp </Content>
contentdef   → <ContentDef (ID=AttValue | RenewID=AttValue)>
                 doc contentexp
```

</ContentDef>

contentexp → <Sequence> (*doc contentexp*)$*$ </Sequence> |
<Optional> *doc contentexp* </Optional> |
<ZeroOrMore> *doc contentexp* </ZeroOrMore> |
<OneOrMore> *doc contentexp* </OneOrMore> |
<Union> (*doc contentexp*)$*$ </Union> |
<AnyElement/> |
<Empty/> |
<If> *doc boolexp*
<Then> *doc contentexp* </Then>
(<Else> *doc contentexp* </Else>)?
</If> |
stringtype |
elementdescr |
contentdescr

A content description that is a reference is evaluated according to Section 2.3. Evaluation of a content description that is not a reference is defined as follows.

The *element names* of a content expression is the set of element names of element declarations in the content expression. An occurrence of *stringtype*, which represents chardata, is treated as a *pseudo-element* declaration, that is, as a declaration of an element with a name different from all other names in the DSD (but one that the same for all occurrences of *stringtype*). Additionally, if the content expression contains an <AnyElement/> or an <Empty/>, then every possible element is considered declared (except the chardata pseudo-element, which can only be declared by an occurrence of *stringtype*).

With *stringtype* regarded as a chardata pseudo-element descriptor, we identify content with an element sequence. A content expression induces for any content a subsequence, called the *projected content*, that consists of the element occurrences whose names are declared by the expression.

Content expression evaluation is carried out by a process of *tentative evaluation*. A successful tentative evaluation defines a *consumed* prefix of the content, along with a *remaining* suffix.

A content expression is evaluated from left to right on the projected original content. The evaluation succeeds if the tentative evaluation succeeds and consumes all of the projected content. In that case, the content expression is said to have been *declared*. Since tentative evaluation is carried out on a subsequence of the children of the original content, it is possible to associate a *current child node* to tentative evaluation on non-empty content: it is the node that corresponds to the first element of the remaining suffix of the content. During tentative evaluation, element defaults are sometimes inserted in front of the remaining suffix. If tentative evaluation of a content expression fails, then all defaults inserted during the tentative evaluation of the expression are removed.

Adjacent chardata nodes (including chardata nodes only separated by comments or processing instructions) are here considered as one node with their content being concatenated.

Tentative evaluation of a content expression on content takes place as follows:

`<Sequence>` $c_1 \ldots c_n$ `</Sequence>` If $n = 0$ then tentative evaluation succeeds and no content is consumed. Now assume that $n > 0$. Then, c_1 is tentatively evaluated on the content. If this evaluation succeeds, then the remaining content is tentatively evaluated with respect to `<Sequence>` $c_2 \ldots c_n$ `</Sequence>`. Otherwise, evaluation fails.

`<Optional>` c `</Optional>` This construct is equivalent to `<Union>` c `<Sequence/>` `</Union>`.

`<ZeroOrMore>` c `</ZeroOrMore>` First, c is tentatively evaluated. If this evaluation consumes content, then the result is the tentative evaluation of `<ZeroOrMore>` c `</ZeroOrMore>` on the remaining content. Otherwise, if no content was consumed, the tentative evaluation of the whole expression succeeds and consumes no content.

`<OneOrMore>` c `</OneOrMore>` This expression is evaluated as `<Sequence>` c `<ZeroOrMore>` c `</ZeroOrMore>` `</Sequence>`.

`<Union>` $c_1 \ldots c_n$ `</Union>` Content expressions $c_1, \ldots, c_n$ are tentatively evaluated starting from the same current child node, in turn, until an evaluation succeeds. Then the evaluation of the whole expression succeeds. If no tentative evaluation succeeds, then the evaluation of the whole expression fails and nothing is consumed.

`<AnyElement/>` This expression succeeds if the content is non-empty in which case it consumes the first element; otherwise, it fails. Note that `<AnyElement/>` does *not* assign an ID to the consumed element.

`<Empty/>` This expression succeeds if the remaining suffix is empty; otherwise, it fails. No content is consumed.

`<If>` b `<Then>` c_1 `</Then>` (`<Else>` c_2 `</Else>`)? `</If>` It is an evaluation error if b is not a meaningful *boolexp*. If b is true, then c_1 is tentatively evaluated. If b is false and c_2 is present, then c_2 is tentatively evaluated. Otherwise, tentative evaluation succeeds and nothing is consumed.

stringtype If the remaining suffix is non-empty and the current child node is a chardata pseudo-element which matches the *stringtype* according to Section 2.13, the tentative evaluation consumes the current child node and succeeds. Otherwise, if an applicable default chardata exists, the selected default is inserted in front of the remaining suffix (as a part of the consumed prefix), and evaluation succeeds. Otherwise, tentative evaluation fails.

elementdescr The tentative evaluation succeeds if the remaining suffix is non-empty, the current child node is a non-pseudo element, and the element description defines a name (Section 2.6) that is the same as that of the current child node. In this case, the tentative evaluation consumes the current child node, and it *assigns* this element the ID of the element description. Otherwise, if

- an applicable default for the name of the element description exists;
- the element description has a `Defaultable` property with value "`yes`" or "`Yes`"; and
- the current node has not itself been inserted as a part of a default

then the default element (including its content) of the selected default is inserted in front of the remaining suffix (as a part of the consumed prefix), it is assigned the ID of the element description, and the evaluation succeeds. Otherwise, tentative evaluation fails.

Note that tentative evaluation of an *elementdescr* does *not* include the evaluation defined in Section 2.6—tentative evaluation only assigns an ID to the node.

contentdescr This content expression is tentatively evaluated as the content expression it contains.

Example

Since DSDs may constrain character data in element content, it is easy to characterize say tables in relational databases:

```
<!-- http://www.brics.dk/DSD/examples/1.0/relational.dsd -->
<DSD IDRef="prices" DSDVersion="1.0">

  <Title>Price tables</Title>

  <ElementDef ID="prices">
    <OneOrMore>
      <Sequence>
        <Element Name="name"/>
        <StringType IDRef="Name"/>
        <Element Name="no"/>
        <StringType IDRef="ProductNumber"/>
        <Element Name="price"/>
        <StringType IDRef="DollarAmount"/>
      </Sequence>
    </OneOrMore>
  </ElementDef>
  ...
</DSD>
```

where `Name`, `ProductNumber`, `DollarAmount` are appropriate string types. A conforming application document may look like

```
<?dsd URI="http://www.brics.dk/DSD/examples/1.0/relational.dsd"?>

<prices>
  <name/>Elbow Joint    <no/>a9382 <price/>23.04
  <name/>Straight Joint <no/>c383  <price/>14.55
</prices>
```

Example

The `audio_content` of the PhoneML language is text (character data) interspersed with `em` elements that themselves contain `audio_content`:

```
<ContentDef ID="audio_content">
  <ZeroOrMore><Content IDRef="audio_item"/></ZeroOrMore>
</ContentDef>

<ContentDef ID="audio_item">
  <Union>
    <StringType/>
    <Element Name="em"><Content IDRef="audio_content"/></Element>
  </Union>
</ContentDef>
```

2.11 Boolean formulas

A *boolean formula* is a *boolean expression*, which is made out of usual boolean connectives and atomic propositions that are attribute descriptions or context patterns.

boolformula → `<Bool` (`IDRef=`*AttValue* `|` `CurrIDRef=`*AttValue*)`/>` `|`
`<Bool>` *doc boolexp* `</Bool>`

booldef → `<BoolDef` (`ID=`*AttValue* `|` `RenewID=`*AttValue*)`>`
doc boolexp
`</BoolDef>`

boolexp → `<And>` (*doc boolexp*)∗ `</And>` `|`
`<Or>` (*doc boolexp*)∗ `</Or>` `|`
`<OneOf>` (*doc boolexp*)∗ `</OneOf>` `|`
`<Not>` (*doc boolexp*)∗ `</Not>` `|`
`<Imply>` *doc boolexp doc boolexp* `</Imply>` `|`
`<Equiv>` (*doc boolexp*)∗ `</Equiv>` `|`
attributedescr `|`
contextpattern `|`
boolformula

A boolean formula that is a reference is explained in Section 2.3; if not a reference, the formula is explained in terms of its boolean expression.

A *boolexp* is *meaningful* (at a point during evaluation of the current element) if all contained attribute descriptions are meaningful (see Section 2.9). A meaningful boolean expression is true according to the following conditions:

`<And>` $b_1 \ldots b_n$ `</And>` Each b_i is true.

`<Or>` $b_1 \ldots b_n$ `</Or>` At least one b_i is true.

`<OneOf>` $b_1 \ldots b_n$ `</OneOf>` Exactly one b_i is true.

`<Not>` $b_1 \ldots b_n$ `</Not>` Some b_i is false.

`<Imply>` b_1 b_2 `</Imply>` Either b_1 is false or b_2 is true (or both).

<Equiv> $b_1 \ldots b_n$ </Equiv> Either each b_i is true or each b_i is false.

attributedescr See Section 2.9.

contextpattern See Section 2.12.

boolformula The formula is true.

Note that evaluation of a boolean expression has no side effects except for those explained in Section 2.3, which details the meaning of circular references.

2.11.1 Specificity of boolean expressions

The *specificity* of a boolean expression is the maximum specificity of any context expression (see Section 2.12.1) that is part of the boolean expression; if there are no such expressions, then its specificity is minimum.

2.12 Context patterns

A *context pattern* determines a set of matching paths in the XML tree. A context pattern is specified as a *context expression*, which is a sequence of *context term* according to which a matching path may be decomposed.

contextpattern	→	`<Context` (`IDRef=`*AttValue* `|` `CurrIDRef=`*AttValue*)`/>` `|`
		`<Context>` *contextexp* `</Context>`
contextdef	→	`<ContextDef` (`ID=`*AttValue* `|` `RenewID=`*AttValue*)`>`
		doc contextexp
		`</ContextDef>`
contextexp	→	(*doc contextterm*)∗
contextterm	→	`<SomeElements/>` `|`
		elementpattern `|`
		contextpattern
elementpattern	→	`<Element` (`IDRef=`*AttValue* `|` `Name=`*AttValue*)?`>`
		(*doc attributedescr*)∗
		`</Element>`

A context pattern that is a reference is explained in Section 2.3; if not a reference, the context pattern is explained in terms of its context expression.

Context expressions and terms *match* paths of elements. Each such element is the current element or is above it.

- A path is matched by a context expression consisting of terms $t_1 \ldots t_n$ if the path may be decomposed into consecutive fragments $p_1 \ldots p_n$ such that p_i is matched by context term t_i for each i.
- A path is matched by an *elementpattern* if it consists of one element node
  - whose name matches the `Name` property, if a such is specified;

  - whose assigned element ID matches the value of the IDRef property, if a such is specified; and
  - each attribute description names a declared attribute in the element, and each attribute description is true.

- Every path is matched by `<SomeElements/>`.
- A path a matched by a `Context` if it is matched by the context expression it contains.

An element node is *matched* by a context pattern if it is the last node in a path that is matched. As a boolean expression, the context pattern is then true; otherwise, it is false.

2.12.1 Specificity of context patterns

Context patterns are compared as follows. Context pattern c_1 has higher *specificity* than c_2 if

1. the number of attribute names in c_1 is larger than that of c_2; or
2. the number of attribute names in c_1 is the same as that of c_2, and the number of element names in c_1 is larger than that of c_2;

Example

With DSDs, we can reconstruct some of the CSS functionality for HTML according to the visual formatting model. In our framework, there is no separate universe of concepts called *properties*; instead, font families, sizes, etc. are represented as attributes. We can even extend CSS selectors, so that they allow selectors to act on font attributes themselves. For example, we may specify that fonts set to 8pt are rendered using the Terminal8 font. To do so, we design the HTML DSD so that all elements specifying font attributes declare the size attribute before the font family attribute. (This could easily be accomplished by defining a constraint `font_constraint` that declares all the font related attributes in a desired order; then, a definition of an element may reference this constraint in order to declare font attributes.) Then, to render 8pt fonts using Terminal8, we would write

```
<Default>
  <Context>
    <Element><Attribute Name="font-size" Value="8pt"/></Element>
  </Context>
  <DefaultAttribute Name="font-family" Value="Terminal8"/>
</Default>
```

To specify that all `h1` headers are rendered in blue, we write

```
<Default>
  <Context><Element Name="h1"/><SomeElements/></Context>
  <DefaultAttribute Name="font-color" Value="blue"/>
</Default>
```

Note that we specify that not only the `h1` elements receive the default value `blue`, but also all their descendants. In this way, we explicitly express that the font color property of the CSS formatting model is inherited.

2.13 String types

A *string type* defines a set of character strings in terms of a regular expression. String types are used to specify valid chardata and attribute values.

stringtype → `<StringType` (`IDRef=`*AttValue* `|` `CurrIDRef=`*AttValue*)?`/>` `|`
`<StringType>` *doc stringtypeexp* `</StringType>`

stringtypedef → `<StringTypeDef` (`ID=`*AttValue* `|` `RenewID=`*AttValue*)`>`
doc stringtypeexp
`</StringTypeDef>`

stringtypeexp → `<Sequence>` (*doc stringtypeexp*)∗ `</Sequence>` `|`
`<Optional>` *doc stringtypeexp* `</Optional>` `|`
`<ZeroOrMore>` *doc stringtypeexp* `</ZeroOrMore>` `|`
`<OneOrMore>` *doc stringtypeexp* `</OneOrMore>` `|`
`<Union>` (*doc stringtypeexp*)∗ `</Union>` `|`
`<Intersection>` (*doc stringtypeexp*)∗ `</Intersection>` `|`
`<Complement>` *doc stringtypeexp* `</Complement>` `|`
`<Repeat Value="`*Numeral*`">` *doc stringtypeexp* `</Repeat>` `|`
`<Empty/>` `|`
`<String Value=`*AttValue*`/>` `|`
`<CharSet Value=`*AttValue*`/>` `|`
`<CharRange Start="`*Char*`" End="`*Char*`"/>` `|`
`<AnyChar/>` `|`
stringtype

A string type or a string type expression defines a regular language over a Unicode alphabet. The alphabet is determined as the declared encoding of the DSD document.

A string type that is a reference is explained in Section 2.3. If it is not a reference and no string type expression is provided, the language defined is the set of all strings. If it is not a reference but a string type expression is provided then the language defined is that of the expression.

The languages of string type expressions are defined as follows:

`<Sequence>` $s_1 \ldots s_n$ `</Sequence>` The concatenation of the languages defined by $s_1, \ldots, s_n$.

`<Optional>` s `</Optional>` The union of the language defined by s and the language containing only the empty string.

`<ZeroOrMore>` s `</ZeroOrMore>` The concatenation of zero or more strings from the language denoted by s.

`<OneOrMore>` s `</OneOrMore>` The concatenation of one or more strings from the language denoted by s.

`<Union>` $s_1 \ldots s_n$ `</Union>` The union of the languages denoted by $s_1, \ldots, s_n$.

`<Intersection>` $s_1 \ldots s_n$ `</Intersection>` The intersection of the languages denoted by $s_1, \ldots, s_n$.

`<Complement>` s `</Complement>` The complement of the language denoted by s.

`<Repeat Value="`n`">` s `</Repeat>` The concatenation of n strings from the language denoted by s.

`<Empty/>` The empty language.

`<String Value="`v`"/>` The singleton set consisting of the string v.

`<CharSet Value="`v`"/>` The set of strings of length one that contain one of the characters in v.

`<CharRange Start="`c_1`" End="`c_2`"/>` The set of strings consisting of a single character whose code is greater than or equal to the code of the character c_1 and less or equal to the code of the character c_2.

`<AnyChar/>` The set of strings of length one.

stringtype The language of the string type.

Note that since characters are specified as attribute values some characters, like '`<`', cannot occur directly, but must be specified by character entities.

A string *satisfies* a string type if it belongs to the language defined by the type.

Example

The definition of `ProductNumber` from the database example in Section 2.10 may look like:

```
<StringTypeDef ID="ProductNumber">
  <Sequence>
    <CharRange Start="a" End="c"/>
    <OneOrMore>
      <CharRange Start="0" End="9"/>
    </OneOrMore>
  </Sequence>
</StringTypeDef>
```

This regular expression would usually be written something like `[a-c][0-9]+`.

2.14 Common syntactic constructs

From the XML specification [8], we adopt the definitions of the following syntactic categories: *Char*, *AttValue*, *CharData*, *element*, and *content*. A *YesOrNo* is one of the strings "`yes`", "`Yes`", "`no`", or "`No`". A *Numeral* is a non-empty string of digits.

2.15 Checking points-to requirements

The points-to requirements are satisfied if for all attributes whose declarations are of type `IDRef`, `RenewID`, or `CurrIDRef`, and that have a `PointsTo` boolean expression, the expression is meaningful and true of the referenced element. This requirement must hold of the document after its evaluation.

2.16 Validity, conformance, and result document

A main DSD is *valid* if all validity requirements in this section hold. An application document is *conforming* if

- it contains a DSD processing instruction specifying a valid main DSD;
- the evaluation of the root element description on the root element succeeds; and
- all points-to requirements are satisfied.

For a conforming document, the *result* of the application document processing is an XML document that is a textual representation of the original XML tree augmented with inserted values of attributes and elements.

The result may also include various information calculated during the processing. This information is specified using a namespace with prefix `DSD` and name `http://www.brics.dk/DSD`. As an example, the element ID assigned to a node in the application document may be added as the value of an attribute `DSD:IDRef`. This parsing information can be useful in subsequent processing by other tools. Similarly, when a DSD processor determines that an application document is not conformant, error messages may be placed inside `DSD:Error` elements added to the tree.

A A DSD for DSDs

The meta-DSD presented here is *complete* in the sense that a document is a *valid* DSD if and only if it conforms to the meta-DSD.

```
<?xml version="1.0"?>
<?dsd URI="http://www.brics.dk/DSD/dsd.dsd"?>

<DSD IDRef="maindsd" DSDVersion="1.0">

  <Title>DSD for DSDs</Title>
  <Version>1.0</Version>
  <Author>Nils Klarlund, Anders Moeller, and Michael I. Schwartzbach</Author>

  <Doc>
    This document is a valid DSD that describes DSD validity.

    For more information about DSD, see the DSD home page at
    <a href="http://www.brics.dk/DSD/">http://www.brics.dk/DSD/</a>.
  </Doc>

  <!-- GENERAL CONCEPTS -->

  <ElementDef ID="maindsd" Name="DSD">
    <AttributeDecl Name="IDRef" IDType="IDRef" Optional="no">
      <PointsTo><Context><Element Name="ElementDef"/></Context></PointsTo>
    </AttributeDecl>
    <AttributeDecl Name="DSDVersion"><String Value="1.0"/></AttributeDecl>
    <Content IDRef="dsdcontent"/>
  </ElementDef>

  <ElementDef ID="subdsd" Name="DSD">
    <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
      <PointsTo><Context><Element Name="ElementDef"/></Context></PointsTo>
    </AttributeDecl>
    <AttributeDecl Name="DSDVersion"><String Value="1.0"/></AttributeDecl>
    <Content IDRef="dsdcontent"/>
  </ElementDef>

  <ContentDef ID="dsdcontent">
    <Sequence>
      <Optional><Element Name="Title"><Content IDRef="content"/></Element></Optional>
      <Optional><Element Name="Version"><Content IDRef="content"/></Element></Optional>
      <Optional><Element Name="Author"><Content IDRef="content"/></Element></Optional>
      <ZeroOrMore>
        <Sequence>
          <Content IDRef="doc"/>
          <Union>
            <Element IDRef="subdsd"/><Element IDRef="default"/><Content IDRef="structdef"/>
          </Union>
        </Sequence>
      </ZeroOrMore>
    </Sequence>
  </ContentDef>

  <ContentDef ID="structdef">
    <Union>
```

```
    <Element IDRef="elementdef"/>
    <Element IDRef="constraintdef"/>
    <Element IDRef="attributedecldef"/>
    <Element IDRef="contentdef"/>
    <Element IDRef="booldef"/>
    <Element IDRef="contextdef"/>
    <Element IDRef="stringtypedef"/>
  </Union>
</ContentDef>

<ContentDef ID="doc">
  <Sequence>
    <Optional><Element Name="Label"><Content IDRef="content"/></Element></Optional>
    <Optional><Element Name="BriefDoc"><Content IDRef="content"/></Element></Optional>
    <Optional><Element Name="Doc"><Content IDRef="content"/></Element></Optional>
  </Sequence>
</ContentDef>

<!-- DEFAULTS -->

<ElementDef ID="default" Name="Default">
  <Optional><Content IDRef="boolexp"/></Optional>
  <ZeroOrMore>
    <Union><Element IDRef="defattribute"/><Element IDRef="defcontent"/></Union>
  </ZeroOrMore>
</ElementDef>

<ElementDef ID="defattribute" Name="DefaultAttribute">
  <AttributeDecl Name="Name"/>
  <AttributeDecl Name="Value"/>
</ElementDef>

<ElementDef ID="defcontent" Name="DefaultContent">
  <Union><Content IDRef="element"/><StringType/></Union>
</ElementDef>

<!-- ELEMENT DESCRIPTIONS -->

<ElementDef ID="elementdescr" Name="Element">
  <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
    <PointsTo><Context><Element Name="ElementDef"/></Context></PointsTo>
  </AttributeDecl>
  <AttributeDecl Name="Name" Optional="yes"/>
  <AttributeDecl Name="Defaultable" Optional="yes">
    <StringType IDRef="YesOrNo"/>
  </AttributeDecl>
  <OneOf><Attribute Name="IDRef"/><Attribute Name="Name"/></OneOf>
  <If><Attribute Name="Name"/><Then><Content IDRef="constraintexp"/></Then></If>
</ElementDef>

<ElementDef ID="elementdef" Name="ElementDef">
  <AttributeDecl Name="ID" IDType="ID"/>
  <AttributeDecl Name="Name" Optional="yes"/>
  <AttributeDecl Name="Defaultable" Optional="yes">
    <StringType IDRef="YesOrNo"/>
  </AttributeDecl>
  <Content IDRef="constraintexp"/>
```

```
</ElementDef>

<!-- CONSTRAINTS -->

<ElementDef ID="constraint" Name="Constraint">
  <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
    <PointsTo><Context><Element Name="ConstraintDef"/></Context></PointsTo>
  </AttributeDecl>
  <AttributeDecl Name="CurrIDRef" IDType="CurrIDRef" Optional="yes">
    <PointsTo><Context><Element Name="ConstraintDef"/></Context></PointsTo>
  </AttributeDecl>
  <Not><And><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></And></Not>
  <If>
    <Not><Or><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></Or></Not>
    <Then><Content IDRef="constraintexp"/></Then>
  </If>
</ElementDef>

<ElementDef ID="constraintdef" Name="ConstraintDef">
  <AttributeDecl Name="ID" IDType="ID" Optional="yes"/>
  <AttributeDecl Name="RenewID" IDType="RenewID" Optional="yes">
    <PointsTo><Context><Element Name="ConstraintDef"/></Context></PointsTo>
  </AttributeDecl>
  <OneOf><Attribute Name="ID"/><Attribute Name="RenewID"/></OneOf>
  <Content IDRef="constraintexp"/>
</ElementDef>

<ContentDef ID="constraintexp">
  <ZeroOrMore>
    <Sequence>
      <Content IDRef="doc"/>
      <Content IDRef="constraintterm"/>
    </Sequence>
  </ZeroOrMore>
</ContentDef>

<ContentDef ID="constraintterm">
  <Union>
    <!-- NOTE: condconstraint must be before contentexp
         to resolve If ambiguity correctly -->
    <Element IDRef="attributedecl"/>
    <Content IDRef="condconstraint"/>
    <Content IDRef="contentexp"/>
    <Content IDRef="boolexp"/>
    <Element IDRef="constraint"/>
  </Union>
</ContentDef>

<ContentDef ID="condconstraint">
  <Element Name="If">
    <Sequence>
      <Content IDRef="boolexp"/>
      <Element Name="Then"><Content IDRef="constraintexp"/></Element>
      <Optional>
        <Element Name="Else"><Content IDRef="constraintexp"/></Element>
      </Optional>
    </Sequence>
```

```
  </Element>
</ContentDef>

<!-- ATTRIBUTE DECLARATIONS -->

<ElementDef ID="attributedecl" Name="AttributeDecl">
  <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
    <PointsTo><Context><Element Name="AttributeDeclDef"/></Context></PointsTo>
  </AttributeDecl>
  <AttributeDecl Name="CurrIDRef" IDType="CurrIDRef" Optional="yes">
    <PointsTo><Context><Element Name="AttributeDeclDef"/></Context></PointsTo>
  </AttributeDecl>
  <Not><And><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></And></Not>
  <If>
    <Not><Or><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></Or></Not>
    <Then>
      <AttributeDecl Name="Name"/>
      <Constraint IDRef="attrdeclattrs"/>
      <Content IDRef="attrdeclcontent"/>
    </Then>
  </If>
</ElementDef>

<ElementDef ID="attributedecldef" Name="AttributeDeclDef">
  <AttributeDecl Name="ID" IDType="ID" Optional="yes"/>
  <AttributeDecl Name="RenewID" IDType="RenewID" Optional="yes">
    <PointsTo><Context><Element Name="AttributeDeclDef"/></Context></PointsTo>
  </AttributeDecl>
  <OneOf><Attribute Name="ID"/><Attribute Name="RenewID"/></OneOf>
  <AttributeDecl Name="Name"/>
  <Constraint IDRef="attrdeclattrs"/>
  <Content IDRef="attrdeclcontent"/>
</ElementDef>

<ConstraintDef ID="attrdeclattrs">
  <AttributeDecl Name="Optional" Optional="yes">
    <StringType IDRef="YesOrNo"/>
  </AttributeDecl>
  <AttributeDecl Name="IDType" Optional="yes">
    <StringType IDRef="IDType"/>
  </AttributeDecl>
</ConstraintDef>

<ContentDef ID="attrdeclcontent">
  <Sequence>
    <Optional><Content IDRef="stringtypeexp"/></Optional>
    <If>
      <Attribute Name="IDType">
        <Union>
          <String Value="IDRef"/><String Value="CurrIDRef"/><String Value="RenewID"/>
        </Union>
      </Attribute>
      <Then>
        <Optional>
          <Sequence><Content IDRef="doc"/><Element IDRef="pointsto"/></Sequence>
        </Optional>
      </Then>
```

```
    </If>
  </Sequence>
</ContentDef>

<StringTypeDef ID="IDType">
  <Union>
    <String Value="ID"/><String Value="IDRef"/>
    <String Value="RenewID"/><String Value="CurrIDRef"/>
  </Union>
</StringTypeDef>

<ElementDef ID="pointsto" Name="PointsTo">
  <Content IDRef="boolexp"/>
</ElementDef>

<!-- ATTRIBUTE DESCRIPTIONS -->

<ElementDef ID="attributedescr" Name="Attribute">
  <AttributeDecl Name="Name"/>
  <AttributeDecl Name="Value" Optional="yes"/>
  <If>
    <Not><Attribute Name="Value"/></Not>
    <Then><Optional><Content IDRef="stringtypeexp"/></Optional></Then>
  </If>
</ElementDef>

<!-- CONTENT DESCRIPTIONS -->

<ElementDef ID="contentdescr" Name="Content">
  <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
    <PointsTo><Context><Element Name="ContentDef"/></Context></PointsTo>
  </AttributeDecl>
  <AttributeDecl Name="CurrIDRef" IDType="CurrIDRef" Optional="yes">
    <PointsTo><Context><Element Name="ContentDef"/></Context></PointsTo>
  </AttributeDecl>
  <Not><And><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></And></Not>
  <If>
    <Not><Or><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></Or></Not>
    <Then><Content IDRef="contentexp"/></Then>
  </If>
</ElementDef>

<ElementDef ID="contentdef" Name="ContentDef">
  <AttributeDecl Name="ID" IDType="ID" Optional="yes"/>
  <AttributeDecl Name="RenewID" IDType="RenewID" Optional="yes">
    <PointsTo><Context><Element Name="ContentDef"/></Context></PointsTo>
  </AttributeDecl>
  <OneOf><Attribute Name="ID"/><Attribute Name="RenewID"/></OneOf>
  <Content IDRef="contentexp"/>
</ElementDef>

<ContentDef ID="contentexp">
  <Sequence>
    <Content IDRef="doc"/>
    <Union>
      <Element Name="Sequence">
        <ZeroOrMore><Content IDRef="contentexp"/></ZeroOrMore>
```

```
    </Element>
    <Element Name="Optional"><Content IDRef="contentexp"/></Element>
    <Element Name="ZeroOrMore"><Content IDRef="contentexp"/></Element>
    <Element Name="OneOrMore"><Content IDRef="contentexp"/></Element>
    <Element Name="Union">
      <ZeroOrMore><Content IDRef="contentexp"/></ZeroOrMore>
    </Element>
    <Element Name="AnyElement"/>
    <Element Name="Empty"/>
    <Element Name="If">
      <Sequence>
        <Content IDRef="boolexp"/>
        <Element Name="Then"><Content IDRef="contentexp"/></Element>
        <Optional>
          <Element Name="Else"><Content IDRef="contentexp"/></Element>
        </Optional>
      </Sequence>
    </Element>
    <Element IDRef="stringtype"/>
    <Element IDRef="elementdescr"/>
    <Element IDRef="contentdescr"/>
  </Union>
 </Sequence>
</ContentDef>

<!-- BOOLEAN FORMULAS -->

<ElementDef ID="boolformula" Name="Bool">
  <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
    <PointsTo><Context><Element Name="BoolDef"/></Context></PointsTo>
  </AttributeDecl>
  <AttributeDecl Name="CurrIDRef" IDType="CurrIDRef" Optional="yes">
    <PointsTo><Context><Element Name="BoolDef"/></Context></PointsTo>
  </AttributeDecl>
  <Not><And><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></And></Not>
  <If>
    <Not><Or><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></Or></Not>
    <Then><Content IDRef="boolexp"/></Then>
  </If>
</ElementDef>

<ElementDef ID="booldef" Name="BoolDef">
  <AttributeDecl Name="ID" IDType="ID" Optional="yes"/>
  <AttributeDecl Name="RenewID" IDType="RenewID" Optional="yes">
    <PointsTo><Context><Element Name="BoolDef"/></Context></PointsTo>
  </AttributeDecl>
  <OneOf><Attribute Name="ID"/><Attribute Name="RenewID"/></OneOf>
  <Content IDRef="boolexp"/>
</ElementDef>

<ContentDef ID="boolexp">
  <Sequence>
    <Content IDRef="doc"/>
    <Union>
      <Element Name="And"><ZeroOrMore><Content IDRef="boolexp"/></ZeroOrMore></Element>
      <Element Name="Or"><ZeroOrMore><Content IDRef="boolexp"/></ZeroOrMore></Element>
      <Element Name="OneOf"><ZeroOrMore><Content IDRef="boolexp"/></ZeroOrMore></Element>
```

```
      <Element Name="Not"><ZeroOrMore><Content IDRef="boolexp"/></ZeroOrMore></Element>
      <Element Name="Imply">
        <Sequence><Content IDRef="boolexp"/><Content IDRef="boolexp"/></Sequence>
      </Element>
      <Element Name="Equiv"><ZeroOrMore><Content IDRef="boolexp"/></ZeroOrMore></Element>
      <Element IDRef="attributedescr"/>
      <Element IDRef="contextpattern"/>
      <Element IDRef="boolformula"/>
    </Union>
  </Sequence>
</ContentDef>

<!-- CONTEXT PATTERNS -->

<ElementDef ID="contextpattern" Name="Context">
  <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
    <PointsTo><Context><Element Name="ContextDef"/></Context></PointsTo>
  </AttributeDecl>
  <AttributeDecl Name="CurrIDRef" IDType="CurrIDRef" Optional="yes">
    <PointsTo><Context><Element Name="ContextDef"/></Context></PointsTo>
  </AttributeDecl>
  <Not><And><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></And></Not>
  <If>
    <Not><Or><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></Or></Not>
    <Then><Content IDRef="contextexp"/></Then>
  </If>
</ElementDef>

<ElementDef ID="contextdef" Name="ContextDef">
  <AttributeDecl Name="ID" IDType="ID" Optional="yes"/>
  <AttributeDecl Name="RenewID" IDType="RenewID" Optional="yes">
    <PointsTo><Context><Element Name="ContextDef"/></Context></PointsTo>
  </AttributeDecl>
  <OneOf><Attribute Name="ID"/><Attribute Name="RenewID"/></OneOf>
  <Content IDRef="contextexp"/>
</ElementDef>

<ContentDef ID="contextexp">
  <ZeroOrMore>
    <Sequence>
      <Content IDRef="doc"/><Content IDRef="contextterm"/>
    </Sequence>
  </ZeroOrMore>
</ContentDef>

<ContentDef ID="contextterm">
  <Union>
    <Element IDRef="elementpattern"/>
    <Element Name="SomeElements"/>
    <Element IDRef="contextpattern"/>
  </Union>
</ContentDef>

<ElementDef ID="elementpattern" Name="Element">
  <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
    <PointsTo><Context><Element Name="ElementDef"/></Context></PointsTo>
  </AttributeDecl>
```

```
  <AttributeDecl Name="Name" Optional="yes"/>
  <Not><And><Attribute Name="IDRef"/><Attribute Name="Name"/></And></Not>
  <ZeroOrMore>
    <Sequence><Content IDRef="doc"/><Element IDRef="attributedescr"/></Sequence>
  </ZeroOrMore>
</ElementDef>

<!-- STRING TYPES -->

<ElementDef ID="stringtype" Name="StringType">
  <AttributeDecl Name="IDRef" IDType="IDRef" Optional="yes">
    <PointsTo><Context><Element Name="StringTypeDef"/></Context></PointsTo>
  </AttributeDecl>
  <AttributeDecl Name="CurrIDRef" IDType="CurrIDRef" Optional="yes">
    <PointsTo><Context><Element Name="StringTypeDef"/></Context></PointsTo>
  </AttributeDecl>
  <Not><And><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></And></Not>
  <If>
    <Not><Or><Attribute Name="IDRef"/><Attribute Name="CurrIDRef"/></Or></Not>
    <Then><Optional><Content IDRef="stringtypeexp"/></Optional></Then>
  </If>
</ElementDef>

<ElementDef ID="stringtypedef" Name="StringTypeDef">
  <AttributeDecl Name="ID" IDType="ID" Optional="yes"/>
  <AttributeDecl Name="RenewID" IDType="RenewID" Optional="yes">
    <PointsTo><Context><Element Name="StringTypeDef"/></Context></PointsTo>
  </AttributeDecl>
  <OneOf><Attribute Name="ID"/><Attribute Name="RenewID"/></OneOf>
  <Content IDRef="stringtypeexp"/>
</ElementDef>

<ContentDef ID="stringtypeexp">
  <Sequence>
    <Content IDRef="doc"/>
    <Union>
      <Element Name="Sequence">
        <ZeroOrMore><Content IDRef="stringtypeexp"/></ZeroOrMore>
      </Element>
      <Element Name="Optional"><Content IDRef="stringtypeexp"/></Element>
      <Element Name="ZeroOrMore"><Content IDRef="stringtypeexp"/></Element>
      <Element Name="OneOrMore"><Content IDRef="stringtypeexp"/></Element>
      <Element Name="Union">
        <ZeroOrMore><Content IDRef="stringtypeexp"/></ZeroOrMore>
      </Element>
      <Element Name="Intersection">
        <ZeroOrMore><Content IDRef="stringtypeexp"/></ZeroOrMore>
      </Element>
      <Element Name="Complement"><Content IDRef="stringtypeexp"/></Element>
      <Element Name="Repeat">
        <AttributeDecl Name="Value"><StringType IDRef="Numeral"/></AttributeDecl>
        <Content IDRef="stringtypeexp"/>
      </Element>
      <Element Name="AnyChar"/>
      <Element Name="Empty"/>
      <Element Name="String"><AttributeDecl Name="Value"/></Element>
      <Element Name="CharSet"><AttributeDecl Name="Value"/></Element>
```

```
      <Element Name="CharRange">
        <AttributeDecl Name="Start"><StringType IDRef="Char"/></AttributeDecl>
        <AttributeDecl Name="End"> <StringType IDRef="Char"/></AttributeDecl>
      </Element>
      <Element IDRef="stringtype"/>
    </Union>
  </Sequence>
</ContentDef>

<!-- COMMON SYNTACTIC CONSTRUCTS -->

<StringTypeDef ID="Char">
  <AnyChar/>
</StringTypeDef>

<StringTypeDef ID="YesOrNo">
  <Union>
    <String Value="yes"/><String Value="Yes"/><String Value="no"/><String Value="No"/>
  </Union>
</StringTypeDef>

<StringTypeDef ID="Numeral">
  <OneOrMore><CharRange Start="0" End="9"/></OneOrMore>
</StringTypeDef>

<ContentDef ID="element">
  <AnyElement/>
</ContentDef>

<ContentDef ID="content">
  <ZeroOrMore><Union><AnyElement/><StringType/></Union></ZeroOrMore>
</ContentDef>

</DSD>
```

B A more readable default syntax

We have strived to make context descriptions very similar to the selector mechanism of Cascading Style Sheets [4]. The XMLized syntax we use should be complemented with a version of CSS syntax that appeals to application document writers. We already gave one example in Section 2.5. Here is another one that shows how content defaults could be formulated:

```
<DSD:Default>
  <Context><Element/></Context>
  <DefaultAttribute Name="font-weight" Value="bold"/>
</DSD:Default>

<DSD:Default>
   <Context>
      <Element Name="menu">
        <Attribute Name="class" Value="myown">
      </Element>
   </Context>
   <DefaultContent>
```

```
        <prompt>
          please enter your <em>selection</em>!
        </prompt>
      </DefaultContent>
    </DSD:Default>
```

could be written more concisely as

```
  <DSD:Defaults>
     * {font-weight: bold}
     menu.myown {<prompt>please enter your <em>selection</em>!</prompt>}
  </DSD:Defaults>
```

C Availability

An experimental implementation of a DSD processor is available from the DSD home page:

http://www.brics.dk/DSD/

This home page also contains DSD examples and an XSL stylesheet for rendering DSD documents in browsers.

References

[1] Murray Altheim et al. Modularization of XHTML. Technical report, W3C, March 1999. W3C Working Draft, Online at http://www.w3.org/TR/1999/xhtml-modularization-19990406/.

[2] David Beech et al. XML Schema part 1: Structures. Technical report, W3C, May 1999. W3C Working Draft.

[3] Paul V. Biron and Ashok Malhotra. XML Schema part 2: Datatypes. Technical report, W3C, May 1999. World Wide Web Consortium Working Draft.

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

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

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

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

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

[9] James Clark. XSL transformations (XSLT) specification. Technical report, W3C, 1999. W3C Working Draft, Online at http://www.w3.org/TR/WD-xslt.

[10] John Cowan. Itsy bitsy teeny weeny simple hypertext. Online at http://www.ccil.org/˜cowan/XML/ibtwsh.dtd.

[11] John Cowan and David Megginson. XML information set. Technical report, W3C, May 1999. W3C Working Draft, Online at http://www.w3.org/TR/xml-infoset.

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

[13] Dave Raggett. Assertion grammars. Draft, Online at http://www.w3.org/People/Raggett/dtdgen/Docs/, May 1999.