

DSD: A Schema Language for XML

Nils Klarlund, AT&T Labs Research

Anders Møller, BRICS, Aarhus University

Michael I. Schwartzbach, BRICS, Aarhus University

Connections between XML and Formal Methods

XML: a notation for labeled trees,
a lot of technologies built on top

Formal Methods: notations and reasoning principles for
complex systems at an abstract level

Our aim: *show that proven FM technologies are
applicable to the world of XML*

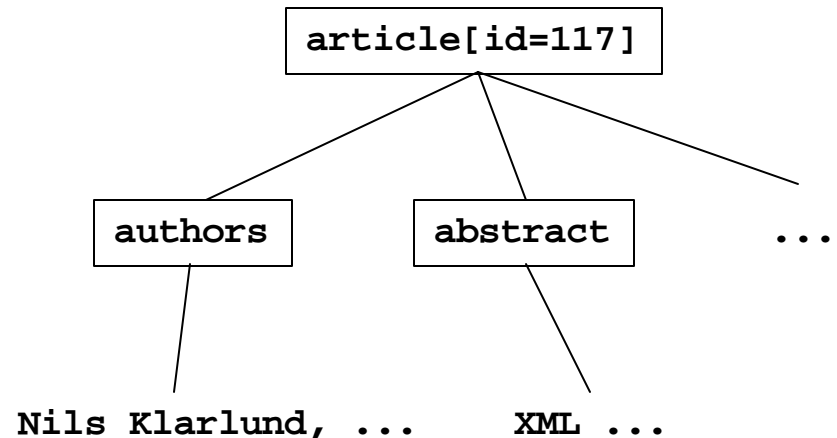
- expressiveness
- simplicity
- efficiency
- elegance

The ideas behind XML

- Textual notation for hierarchically structured information
- Built-in internationalization and platform independence

```
<?xml version="1.0"
      encoding="iso-8859-1">
<article id="117">
  <authors>
    Nils Klarlund, ...
  </authors>
  <abstract>
    XML (eXtensible Markup Language)
    is a linear syntax for ...
  </abstract>
  ...
</article>
```

concrete, textual representation



abstract, tree representation

Central technologies building on top of XML

- Schema languages (grammar notations)
e.g. DTD, **DSD**, XML Schema, ...
- Transformation languages
e.g. XSLT, XDuce, ...
- Query languages
e.g. XML-QL, Quilt, ...
- ...

All could benefit from Formal Methods!

Our DSD proposal: *a schema language with a CS foundation.*

Each DSD document defines (essentially) a *regular tree language*.

DSD: Document Structure Description

What schemas can do for you

- Formalize your notion of “articles” by making a DSD description
- Check validity of your article XML documents using a DSD processor

Example:

1. Insert `<?dsd URI="article.dsd">` in the article header
2. Run the DSD processor: `dsd article.xml`

The result might be:

```
Error in 'article.xml' line 17: attribute 'font' has illegal
value 'ttt' (checking attribute in 'article.dsd' line 84)
```

In order to be useful, the schema notation must

- have enough **expressive power**
- be **efficiently implementable**
- be **easily understandable**

Design goals

DSD must

- allow **linear-time** processing
- be 100% **self-describing**
- subsume the **expressive power** of DTD
- contain a **default mechanism** for attributes, elements, and text
- support **modularization, evolution, and reuse**
- be able to express **semi-structured data**
- handle **syntax for attributes and text**
- support **context dependencies**
- support **conditional constraints**

A quick tour of DSD

Processing model: like a *deterministic top-down automaton*,
but extended with

- default insertion
- a sophisticated transition function with context sensitivity

(A top-down approach complies with the recursive structure of typical XML languages!)

Each node is associated a **state** called an “**Element Definition ID**”, referring to a description in the DSD of a particular kind of element.

If the top-down traversal succeeds, the document “**conforms**” to the DSD.

An example: a DSD for XML “business cards”

An application document:

```
<card type="simple">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 456-1414</phone>
</card>
```

A part of a DSD for business-card XML documents:

```
<DSD IDRef="card-element">
  <Title>This is a DSD for XML business cards</Title>
  <ElementDef ID="card-element" Name="card">
    <AttributeDecl Name="type" Optional="yes">
      <Union><String Value="simple"/><String Value="complex"/></Union>
    </AttributeDecl>
    <Element Name="name"><StringType/></Element>
    <If><Attribute Name="type" Value="simple"/><Then> ... </Then></If>
    ...
  </ElementDef>
  ...
</DSD>
```


Element constraints

The central DSD construct:

```
<ElementDef ID="Element ID" Name="element name">  
  constraint  
</ElementDef>
```

It associates an **element name** and a **constraint** to an **Element Definition ID**.

A **constraint** declares and constrains *attributes* and *content* of the element.

Constraints can be made **conditional** on attributes and context.

Attribute declarations

In a constraint,

- element attributes can be **declared hierarchically**
- their allowed **values** can be constrained

String types

We use standard **regular expressions** to define valid values for **element attributes** and **text**.

All well-known data types can be described by regular expressions:

- URLs
 - email addresses
 - ZIP codes
 - ISBN numbers
 - ...
- and they can be processed efficiently using automata techniques.

Other schema notations rely on fixed sets of data types.

Content expressions

Content of an element: its sequence of child nodes

Obvious choice for specifying valid content: *regular expressions*

However:

- that would conflict with default insertion
- not obvious how to assign a single Element Definition ID to each child element

The DSD solution: a notation resembling regular expressions, but

- having a **direct operational semantics**
- support for both **ordered** and **unordered** contents
- each child node is **matched exactly once**

Context patterns

In practice, many validity constraints on an element depend on its **context**. (Same for Cascading Style-Sheets.)

Element context: the sequence of elements above it in the tree

DSD constraints can

- be conditional on the context
- impose requirements of the context

Default insertion

Insertion of defaults

- is very useful
- must be part of the validation process!

The DSD default mechanism is **declarative**; each default consists of

- an applicability expression (probing context etc.)
- an XML fragment

Defaults are inserted “upon request” during the top-down traversal.

ID attributes and Points-to requirements

- going slightly beyond regularity

Special attributes:

ID attributes: their values must be **unique** in the document

IDRef attributes: their values must be that of some **ID** attribute

- i.e., they can be seen as internal **definitions** and **references**

Additionally, **points-to** requirements can be imposed on ID attributes. This allows ***semi-structured data*** to be expressed.

Redefinitions and evolving DSDs

Modification, extension, and reuse is crucial.

The DSD solution:

- document inclusion (allows reuse)
- selective redefinitions (for modification and extension)

DSD highlights

- Large class of tree languages, handling almost all syntactic constraints of boolean or regular nature
- Grammar modifications: abstraction, restriction, extension
- Context dependencies, semi-structured data description, and defaults

Related work

- **DTD** (part of XML 1.0 spec.)
 - general agreement that it is much too simple
- **RELAX** (by Makoto Murata)
 - elegantly characterizes the set of all regular tree languages
 - uses bottom-up approach
- **XML Schema** (draft specification, by W3C WG)
 - much more complex than other proposals
 - significantly less expressive: no boolean dependencies, schema evolution, context sensitivity, declarative default insertion, semi-structured data, ... (but supports namespaces, inheritance, and uniqueness)
 - has been receiving harsh comments for being convoluted and incomprehensible

Conclusion

XML represents an excellent opportunity to promote formal descriptions.

Future work:

- minor extensions to DSD
- simplify DSD even further, and obtain a cleaner semantics
- apply Formal Methods to other areas of XML

DSD

Document Structure Description

Document Structure Description (DSD) is an XML schema language. A DSD document is a specification of a class of XML documents together with a default mechanism and documentation. The DSD notation is defined in the [DSD 1.0 specification](#). The DSD project is being pursued at [AT&T Labs Research](#) and at [BRICS, University of Aarhus](#) in a non-proprietary fashion.



DSD provides an alternative to XML DTDs and other XML schema languages. It adds expressive power, increases readability, and contains support for default attributes and contents. Furthermore, it guarantees linear time processing in the size of the application document.

The relationship between DSDs and XML Schema is briefly described in [this FAQ](#).

A prototype [DSD processor](#) has been implemented. It is freely available under the GNU Public Licence for experimentation and further development.

The DSD 1.0 language has been designed by [Nils Klarlund](#), [Anders Møller](#), and [Michael I. Schwartzbach](#).

For inquiries, comments, or questions about DSD, please contact dsd@brics.dk.

Available resources:

- [DSD announcement](#)
- [DSD 1.0 specification](#)
- [Overview article](#)
- [DSD description of DSDs](#)
- [Download free source code and Win32 executables](#) NEW!
- [XSLT style sheet](#)
- [DSD industrial case](#)
- [Other examples](#)
- [DSD presentations](#)
- [XML tutorial](#) NEW!
- [Future issues](#)