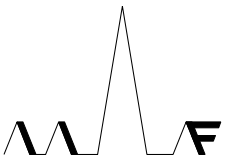
University of Freiburg



Institute for Computer Science

XInterfaces -

A new schema language for XML

Final thesis of

Oliver Nölle

June 12, 2002

University of Freiburg, Germany

Institute for Computer Science

Programming Languages Group

Prof. Dr. Peter Thiemann

Abstract

A new schema language for XML is proposed to enhance the interoperability of applications sharing a common dataset.

An XML document is considered as a semi-structured database, which evolves over time and is used by different applications. An XInterface defines a view of an XML document by imposing constraints on structure and type of selected parts. These constraints are not grammar-based but specify an open-content model, allowing additional elements and attributes to be present anywhere in the document. This enables each application to define and validate its own view on the document, with data being shared between applications or specifically added by one application.

XInterfaces feature an explicit type hierarchy, enabling easy extension of existing schemas and documents while guaranteeing conformance of the extended documents to the existing views. This allows data evolution without breaking compatibility of existing applications. Because different applications share one document, access mechanisms are described that guarantee the validity of the document for all applications after modifications.

As a proof of concept, a tool was implemented that maps XInterfaces to a class framework in Java, allowing convenient access to those parts of an XML document that are described by XInterfaces.

Declaration of originality

This is to certify that

1. the presented material comprises only my original work towards my diploma,
2. due acknowledgment has been made in the text to all other material used,
3. the material has not been accepted for the award of any other degree or diploma.

Hiermit erkläre ich, daß ich diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.
Diese Diplomarbeit wurde nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt.

Freiburg, 12.6.2002

Acknowledgments

I would like to express my full gratitude to my thesis supervisor, Prof. Dr. Peter Thiemann. His close guidance and early feedback was invaluable help for getting me on the way, his criticism was always justified and very constructive - thanks a lot! Thanks to Jenny, Bob and Hardy for proofreading (no "sexy coding" issues this time, though...), to my family and Joe for constant mental support, and to Laura for making me laugh so many times...

Contents

List of Tables

Chapter 1

Introduction

This chapter introduces the Extensible Markup Language and its origins, briefly explains the basic concept of a schema and the motivation to create a new schema language.

1.1 The Extensible Markup Language

The Extensible Markup Language, or XML[8], is a universal format for structured documents and data on the Web. A quick glance at XML's history reveals that it is derived from the Standardized Generalized Markup Language and strongly influenced by the Hypertext Markup Language.

XML's origins in SGML

XML is derived from the Standardized Generalized Markup Language (SGML), whose origins date back to research at IBM in the late 1960s. The General Markup Language (GML) was published 1969 as an attempt to satisfy the need for sharing documents in different text editing, formatting or retrieval systems. The American National Standards Institute (ANSI) published its first version of SGML in 1978, and it finally became an ISO-standard in 1986 [19]. SGML was widely adopted in the document processing and desktop publishing community. However, SGML has the reputation of being too complex to author and to implement, and never made it into the low-end desktop publishing market.

The Internet and HTML

With the Internet emerging and becoming a mass media an application of SGML came into place: The Hypertext Markup Language (HTML). The simplicity of HTML resulted in a short learning curve and thousands of Internet users suddenly became authors of HTML documents. However, HTML imposes a fixed document structure, and some of its markups imply a certain layout semantics. This drawback became more and more obvious as the Internet evolved, and applications such as E-Commerce required software agents to gather and exchange information via the Internet. As opposed to human readers, software agents do not care about layout, but need structural markup to correctly parse and interpret information.

The creation of XML

With SGML being too complex (and also lacking the support of Microsoft) and HTML being not sufficiently flexible while focusing too much on layout, the need for something to fill the gap became urgent, and in 1996 the XML Working Group was founded at the World Wide Web Consortium (W3C). This group was dedicated to "bringing the key benefits of generic SGML to the Web in a manner that is easy to implement and understand while remaining fully compliant with the ISO standard"[18].

The efforts of the XML working group culminated with the release of XML 1.0 as a W3C recommendation (which represents a standard and is the most popular form of standardization for web-related technologies) in February 1998. The simplicity of usage and implementation seemed to encourage the web community to rapidly accept and support the XML standard. Tools and standards quickly emerged around the core XML technology. Although originating from the document markup background, XML was discovered to be equally suited for data centric applications, establishing XML technologies as an alternative to traditional database approaches, in particular in the area of semi-structured data.

Today, four years after the XML standard was first published, XML is *the* standard for information interchange, major technologies are relying on XML, people talk of XML as shaping the fourth computer revolution (following the personal computer, graphical user interfaces and the internet, [34]) and few software products on the market do not include some kind of "XML interoperability" in their feature list.

Here is a fragment of an XML document to illustrate how a document tagged

with markup looks like:

Fragment of an XML document

```
<contact Category="business">
  <Name>Peter Jones</Name>
  <Phone>0172555666</Phone>
  <Address>
    <Street>Private Drive 4</Street>
    <City>London</City>
  </Address>
</contact>
```

1.2 Schema Languages

With the importance of XML, the importance of schema languages grew. Many people considered the feature of configurable document types as one of the strengths of SGML and at the same time the lack of it the main weakness in HTML. A schema defines a type or class of documents, allowing validity checks and an interpretation that goes far beyond the interpretation of an untyped document.

The XML recommendation itself comes with a built-in schema language (Document Type Definition), whose shortcomings very soon resulted in many activities to create new schema languages. As the most recognized successor to DTDs, the W3C published the the first recommendation for XML Schema[39] in May 2001. Document Type Definition, XML Schema and other schema languages are further described in chapter 3.

The following is a fragment of an XML Schema schema. Leaving the details for later, it basically expresses that `contact` elements have to carry a `Category` attribute and that their content consists of first a `Name` and then a `Phone` element, whose content contains an `integer`.

Fragment of a schema

```
<xsd:element name="contact>
  <xsd:complexType>
    <xsd:attribute name="Category" type="xsd:string"/>
    <xsd:sequence>
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="Phone" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Given a complete schema, an application can validate a given XML document, i.e., check, whether the document has the format that is defined in the schema. If the document is valid, the application can rely on a certain structure of the document, e.g., it knows that there is a `Phone` element of type `integer` for every `contact` element.

1.3 A need for a different schema language

While most of the existing schema languages are well suited for describing the structure of one class of documents, the scenario in the next chapter requires a different approach: With the help of XInterfaces it is possible to describe requirements for XML documents, allowing *one* instance document to satisfy the requirements of *many* different XInterfaces.

This distinction resembles the distinction of classes and interfaces in object-oriented programming languages: While most existing schema languages allow powerful mechanisms to exhaustively define the structure of a document (like a class), XInterfaces define a set of properties (like an interface), and XML documents can register themselves as implementing any number of XInterfaces. This approach also enables XInterfaces to support multiple inheritance, a feature which is not available for most existing schema languages[1].

The interface approach allows data sharing from otherwise independent applications, enables data evolution without breaking compatibility and allows seamless integration into a single-inheritance/multiple-interfaces object-oriented programming language such as Java. Its simplicity suggests that this approach is a well-suited alternative (or enhancement) to existing schema languages, in particular when the emphasis is on transparent data sharing.

The following is a very simple XInterface type definition, very similar to the XML Schema schema given above. It can be considered as defining a view on an XML document, requiring that certain elements need to carry a `Category` attribute and `Name` and `Phone` elements as children. Another view could require for the same elements also a `Name` element, but instead of the `Phone` element an `Address` element as content. In contrast to XML Schema, these views allow additional elements and attributes interleaved with the

[1] Multiple inheritance was apparently under consideration for XML Schema, but the working group decided for a strong single inheritance model first [41]. In a personal email to me Henry S. Thompson confirmed that there is "No sign of multiple inheritance in the near future." [42]

required ones, and do not enforce a particular order on the elements.

Sample XInterface

```
<schema>
  <interfaceType  name="MobileContact1">
    <attribute  name="Category" type="string"/>
    <element  name="Name" type="string"/>
    <element  name="Phone" type="integer"
              maxOccurs="5" initial="0"/>
  </interfaceType>
</schema>
```

1.4 Roadmap

Chapter 2 introduces a sample scenario that motivates the requirements for the XInterface language and will be referenced in the following chapters.

Chapter 3 gives a short overview of existing schema languages, focusing on their shortcomings when applied to the given scenario.

Chapter 4 lists the requirements that drove the design of the XInterface language. Most of them are derived from the requirements illustrated in the scenario.

Chapter 5 introduces the XInterface schema language in detail.

Chapter 6 shows the usage of XInterfaces for the sample scenario.

Chapter 7 concludes with a summary, a list of ideas for improvements and a vision outlining another scenario where XInterfaces are a perfect fit.

Appendix A gives XML Schema and XInterface schemas to describe the syntax of XInterface types.

Appendix B shows source codes generated by the class framework generator for the sample scenario.

Chapter 2

Scenario

The following scenario serves as a motivating example throughout the remainder of this thesis and informally illustrates most of the requirements for a new XML schema language. The scenario relies on XML as the data storage format, but is independent of existing schema languages.

A common scenario is a person is using different applications (possibly running on different devices such as notebook, PDA or mobile phone) like a calendar, email client, or other so-called PIM (personal information management) applications. A lot of data that is used by one application is also important for other applications, in particular a "contacts" database could be used by the email client on the PDA and by the mobile phone. Today, separate datasets are often used. This requires synchronization mechanisms as well as import and export possibilities if different applications access these datasets.

Instead of keeping separate datasets on each device/for each application, it is desirable to reuse existing data, thus avoiding redundancies, which in turn avoids inconsistencies. The increasing "connectedness" of all sorts of applications via Internet, wireless LAN or bluetooth and technologies like flash memory enable this sharing and reusing of data from the hardware point of view.

For this thesis we assume the dataset is represented as a file and is readable and writable (e.g., on a flash memory card or via an Internet connection) for all applications.

2.1 Step 1 - Basic dataset

The first step represents a very simple dataset which might be used by a mobile phone for storing its telephone book in XML format. Table 2.1 shows a possible XML document containing two `contact` entries.

Table 2.1: Dataset for scenario, step 1

```
<?xml        version="1.0"?>
<!-- scenario.xml - instance document for the scenario -->

<pimData>

  <contact>
    <Name>Peter Jones</Name>
    <Phone>0172555666</Phone>
    <Address>
      <Street>Private Drive 4</Street>
      <City>London</City>
    </Address>
  </contact>

  <contact>
    <Name>Lucy Walsh</Name>
    <Phone>0179445566</Phone>
    <Address>
      <Street>Parkstr. 7</Street>
      <City>Munich</City>
    </Address>
    <Address>
      <Street>Bahnstr. 8</Street>
      <City>Freiburg</City>
    </Address>
  </contact>

</pimData>
```

Given a suitable schema language, an application should be able to validate the dataset according to a given schema, e.g., to ensure that

- all `contact` elements provide a `Name` and a `phone` element (**structure**)
- all `contact` elements provide exactly one `Name` element, but up to five `Address` elements (**cardinality**)

- the content of all `Phone` elements has the appropriate type, i.e. only containing digits (**typing**)

2.2 Step 2 - Data sharing

A common scenario will be another application sharing the dataset and storing additional information. For example, the email client might need an email address and an optional nickname as well as a category attribute for each contact, and shares the `Name` element. It is not problematic to simply add elements at the appropriate locations to the XML document, so the extended first contact element could look like shown in table 2.2.

Table 2.2: Extended `contact` element for scenario, step 2

```
<contact  Category="business">
  <Name>Peter Jones</Name>
  <Phone>0172555666</Phone>
  <Address>
    <Street>Private Drive 4</Street>
    <City>London</City>
  </Address>
  <Nickname>PJ</Nickname>
  <Email>peter.jones@web.de</Email>
</contact>
```

The fact that two applications access the same dataset should not create any conflicts, each application should be guaranteed its own "view" on the data. For read access to elements this means ignoring irrelevant parts, for write access it should be ensured that validity of the dataset is not lost when elements are changed, added or deleted.

2.3 Step 3 - Data evolution

Even for a single application a defined data format is unlikely to stay fixed for a long time, as not all future developments can be anticipated. As a consequence, additions to a data format which result from data evolution should be possible in a straightforward way, using the potential of existing formats. Therefore a mechanism for extending existing schemas is desirable.

A new mobile phone might be used which is capable of accessing web pages and offers to store one or more URLs for each contact, anticipating that more and more contacts will be associated with internet sites. For this new application the dataset needs to be extended with an additional homepage field. Instead of creating a new schema it should be based on the old one, expressing that the new schema is basically the old one plus some additional constraints. In particular, every application designed to work with data conforming to the old schema (e.g., the old mobile phone) should function properly with the extended dataset as well, ensuring backward compatibility of the dataset.

2.4 Step 4 - Name conflicts

As the dataset is not restricted in any way and a single application does not know which elements might be added in future for other applications, naming conflicts cannot be avoided.

For example, the email client could use the `Category` element for a categorization into business or private contacts. That is, the element content could be either "business" or "private". At the same time, the mobile phone might also make use of a `Category` element, using a number from 0 to 9 to classify the contacts according to priority. As the two `Category` elements represent different semantic concepts they cannot be shared (as the `Name` element was shared between the different applications), but have to be distinguished in the instance document.

The resulting instance document which uses namespaces for name conflict resolution and has additional markup, which will be explained in chapter 5, is given in section 6.1.

2.5 Graphical summary

Figure 2.1 summarizes the resulting data structure in terms of a UML-style[22] class diagram. Each "view" that an application defines on the data by specifying a set of requirements is represented as a class. The extension of existing requirements as in step 3 is modeled as derived classes. The diagram also includes a class "Contact" which is derived from all other classes and therefore represents the resulting requirements for an element conforming to all views.

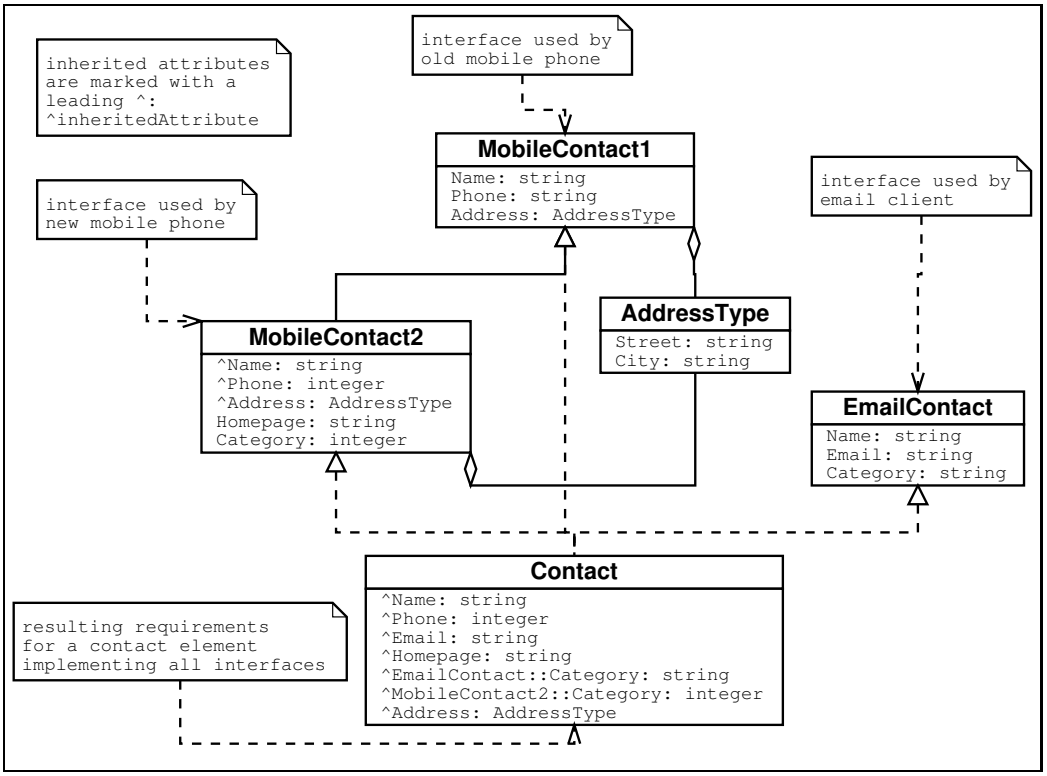


Figure 2.1: Data structure of the scenario

Chapter 3

Existing schema languages

3.1 Introduction

Reflecting the importance of XML as *the* standard for data exchange, much effort was put into the investigation of schema languages that allow the definition of a class of documents. This chapter investigates existing schema languages and tries to apply these languages to the sample scenario. The apparent shortcomings illustrate the requirements for the XInterface language.

As comparing all existing schema languages in detail is beyond the scope of this thesis, the focus is on the most popular and widely used languages. For further details see the given references or available comparisons of schema languages, such as Lee and Chu[32], who include a comprehensive list of supported features for six schema languages, or van der Vlist[44], who also summarizes the history of the major schema languages.

There does not seem to be a single commonly accepted approach to categorize schema languages. Many classifications are based on the support of specific features such as inheritance mechanisms or data types. Figure 3.1, a slightly modified and updated version of a diagram originally provided by Jelliffe[26], provides a family tree of XML schema languages, depicting the evolution of the major schema languages within the last years. The diagram roughly orders schema languages on an axis according to whether their approach is rather grammar-based or relies on patterns and constraints.

The schema languages with a bold border are the ones that are investigated in the next sections.

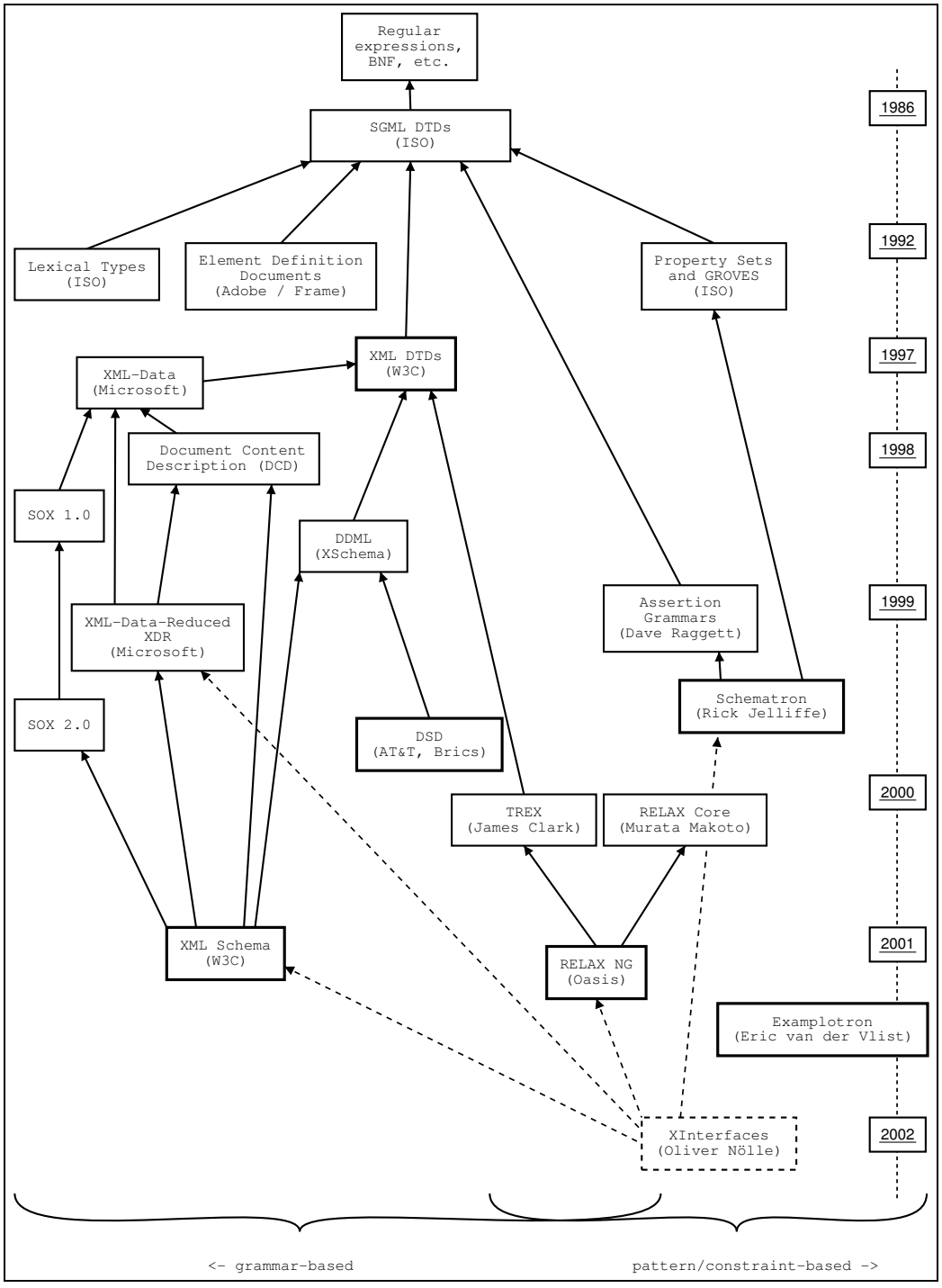


Figure 3.1: Family tree of schema languages

3.2 Document Type Definition (DTD)

This schema language was quickly adopted by the web community and the industry, because the specification of Document Type Definitions was included in the XML 1.0 specification [8]. However, several important shortcomings led to the development of many new schema languages in the last few years. Nevertheless, DTDs are still widely used today.

XML DTD is a subset of SGML DTD. Its main building blocks consist of element and attribute declarations. DTD does not support namespaces, which were introduced later. The lack of support for namespaces and the weak type system which only applies to attributes were the main motivation for the W3C to develop a new schema language. Moreover, DTD is one of the few schema languages that does not use XML syntax, another drawback that most alternatives are addressing.

Shortcomings

Table 3.1 shows a possible DTD for the data from step 1 of the sample scenario.

Table 3.1: DTD for scenario, step 1

```
<!-- DTD for scenario, step1 -->

<!ELEMENT pimData (contact*)>
<!ELEMENT contact (Name, Phone, Address+)>

<!ELEMENT Address (Street, City)>

<!ELEMENT Name (#PCDATA)>
<!ELEMENT Phone (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
```

The instance document has to refer to this DTD with a corresponding document type declaration like this:

```
<!DOCTYPE pimData SYSTEM "contacts.dtd">
```

The DTD cannot supply type information for elements and it cannot express precise cardinalities in a straightforward way. When advancing to step 2 of the scenario more problems become obvious: The schema defined by this DTD does not support an open-content model, i.e., any additions to the instance document which are not specified in the DTD invalidate the document with respect to the given schema. At the same time, the document type declaration binds the instance document to exactly one DTD, making it impossible to express that one document adheres to multiple schemas.

3.3 XML Schema

XML Schema [39] is the official schema language from the W3C which is intended to eventually replace DTDs. The XML Schema Working Group considered several submitted proposals for XML-based schema languages and published a requirements document in early 1999. The specification was finally published in May 2001 as a W3C recommendation. Considered proposals among others were XML-Data[31], XML-Data-Reduced [20], DCD (Document Content Description for XML)[6], SOX (Schema for Object-Oriented XML) [15], DDML (Document Definition Markup Language or XSchema)[5].

XML Schema includes an advanced type system and support for type inheritance (by restriction and by extension), uses XML syntax, and allows flexible content models. XML Schema is widely supported, but has the reputation of being complex to use and implement. The specification itself [43] is not only long but very technical and complex (for feedback and criticism on the specification see Robie[38] or Alschuler[1]). Nevertheless, XML Schema addresses all of DTD's shortcomings and by now probably is the schema language best supported by industry and web community.

Shortcomings

Given the sample scenario, table 3.2 shows a possible schema to model the data of step 1.

In comparison with the DTD from table 3.1 there is now type information for element content and the cardinalities can be specified exactly. However, the schema still does not support an open-content model. Thus validation with the same schema fails when the information needed for step 2 is added

Table 3.2: XML Schema schema for scenario, step 1

```
<?xml version="1.0"?>
<!--  scenario_step1.xsd
      XML Schema schema for scenario, step 1 -->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="pimData">
  <xsd:complexType>
    <xsd:sequence>
      <element name="contact" type="ContactType"
               minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="ContactType">
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Phone" type="xsd:integer"/>
    <xsd:element name="Address" type="AddressType"
             minOccurs="1" maxOccurs="5"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="AddressType">
  <xsd:sequence>
    <xsd:element name="Street" type="xsd:string"/>
    <xsd:element name="City" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

to the instance document. Sun's Multi-Schema XML Validator[28] generates the following error:

```
Error at line:11, column:12
  element "Email" was found where no element may occur
```

Although the XML Schema specification contains the notion of "lax validation" and allows the use of "anyType" to accept any well-formed XML to appear within an element, this kind of open-content model is not suited for the requirements of the sample scenario. The problem is that the schema still has to be define *where* unknown content should be allowed and in the scenario this cannot be known in advance[1]. A typical usage of the `anyType` type is allowing well-formed HTML to appear as the content of a particular element. See the example for RELAX NG (which offers the same concept of a declaration matching all elements) to see how this could be used to simulate an open-content model, and what the limitations are.

3.4 RELAX NG

RELAX NG [11] is the result of merging RELAX[33] and TREX[10], which in turn was influenced largely by XDuce[24].

The key features of RELAX NG are "that it is simple, easy to learn, uses XML syntax, does not change the information set of an XML document, supports XML namespaces, treats attributes uniformly with elements so far as possible, has unrestricted support for unordered content, has unrestricted support for mixed content, has a solid theoretical basis, and can partner with a separate datatyping language (such as W3C XML Schema Datatypes)" [44].

Shortcomings

Table 3.3 shows a possible RELAX NG schema for step 1 of the sample scenario.

[1] XML-Data-Reduced features some of the properties required for the scenario, in particular the true open-content model and an extends mechanism allowing the import of existing schemas while guaranteeing the subsumption property. Unfortunately these features did not make it into XML Schema and XML-Data-Reduced was not further developed since its submission as a draft document in July 1998, as Microsoft promised to stop promoting XML-Data-Reduced in favor of a common XML Schema standard.

Table 3.3: RELAX NG schema for scenario, step 1

```
<?xml version="1.0"?>
<!--  scenario_step1.rng
      RELAX NG schema for the scenario, step 1-->

<element  name="pimData"
          xmlns="http://relaxng.org/ns/structure/1.0">

  <zeroOrMore>
    <element name="contact">
      <element name="Name">
        <text/>
      </element>
      <element name="Phone">
        <text/>
      </element>
      <element name="Address">
        <element name="Street">
          <text/>
        </element>
        <element name="City">
          <text/>
        </element>
      </element>
    </element>
  </zeroOrMore>
</element>
```

Similarly to XML Schema, the RELAX NG schema can be modified to simulate an open-content model. This can be done by defining a type `anyElement` which validates against any well-formed XML and by allowing the content model to contain an arbitrary number of elements of this type interleaved with the required elements. Table 3.4 shows such a schema. As the required elements need to be validated against their declaration and should not match the `anyElement` declaration, the schema has to mention their names explicitly as an exception to the `anyElement` declaration. This illustrates the main shortcoming of RELAX NG and XML Schema: The open-content model has to be simulated with additional constructs. This results in unnecessary complex schemas and makes the approach not a very natural solution for the requirements of the scenario.

3.5 Schematron

Schematron [27] was proposed in September 1999 and is based on XPath[12] expressions to define assertions that must be met by instance documents. Its approach differs from other schema languages as it is not based on grammars but on finding tree patterns in the parsed document. This allows many kinds of structures to be represented which are difficult or impossible to represent in grammar-based schema languages.

Schematron inherits the full expressiveness of XPath, at the same time being easy to implement on top of existing XPath implementations. As one of the few non-grammar approaches it can naturally model open-content models and express many integrity constraints that are impossible to specify in XML Schema. For example, the grammar approach is not sufficient to express any constraints between information items in different branches of the attribute-value tree which forms the primary view of an XML document. Schematron has a notion of "usage patterns" which can be individually turned on or off. These usage patterns are conceptually very close to the interface approach. However, Schematron does not feature a type hierarchy or inheritance mechanisms.

Shortcomings

The assertions shown in table 3.5 could be used for step 1 of the scenario.

It is easy to extend the instance document without altering this schema. At the same time another schema can be defined and applied to the same

Table 3.4: RELAX NG schema with simulated open-content model

```
<?xml version="1.0"?>
<!-- RELAX NG schema for the scenario, step 1-->
     modified version to simulate an open-content model -->

<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="pimData"/>
  </start>

  <define name="anyElement">
    <element>
      <anyName>
        <except>
          <name>Name</name>
          <name>Phone</name>
        </except>
      </anyName>
      <zeroOrMore>
        <choice>
          <attribute>
            <anyName/>
          </attribute>
          <text/>
          <ref name="anyElement"/>
        </choice>
      </zeroOrMore>
    </element>
  </define>

  <define name="pimData">
    <element name="pimData">
      <zeroOrMore>
        <element name="contact">
          <interleave>
            <element name="Name">
              <text/>
            </element>
            <element name="Phone">
              <text/>
            </element>
            <zeroOrMore>
              <ref name="anyElement"/>
            </zeroOrMore>
          </interleave>
        </element>
      </zeroOrMore>
    </element>
  </define>
</grammar>
```

Table 3.5: Schematron schema for scenario, step 1

```
<?xml version="1.0"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
  <pattern name="assertions for scenario, step 1">
    <rule context="/pimData/contact">
      <assert test="Name"> Name missing </assert>
      <assert test="Address/Street"> Street missing </assert>
      <assert test="Address/City"> City missing </assert>
      <assert test="Phone"> Phone missing </assert>
    </rule>
  </pattern>
</schema>
```

instance document, e.g., modeling the requirements for step 2 of the sample scenario as shown in table 3.6.

Table 3.6: Schematron schema for scenario, step 2

```
<?xml version="1.0"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
  <pattern name="test for step 2">
    <rule context="/pimData/contact">
      <assert test="Name"> Name missing </assert>
      <assert test="Email"> Email missing </assert>
      <assert test="Nickname"> no Nickname (optional) </assert>
      <assert test="@Category"> Category missing </assert>
    </rule>
  </pattern>
</schema>
```

While Schematron is the first of the investigated schema languages to naturally express the open-content model approach and different Schematron schemas can be applied to the same instance document, the following desired properties are not supported:

1. There should be an extension mechanisms for existing schemas, which guarantees backward compatibility of the dataset. This is important because extending a schema will be a very common step.

2. For applications that want to use the data described by the schema, a type system is not only useful but essential.

3.6 Assertion Grammars

Assertion grammars [16], developed in 1999, feature a concept of tree patterns comparable to that in Schematron, but without XPath expressions as a means to specify a validation context .

Development of Assertion grammars stopped after the first release in 1999. As the expressive power can be mapped to Schematron constructs and the language has limited popularity, it is not investigated in detail.

3.7 Examplotron

Examplotron [45] started from the observation that instance documents are usually much easier to understand than the schemas which are describing them. Instead of giving examples of instance documents to help human readers to understand the schema, Examplotron allows the definition of patterns by example instance documents.

As not all constraints can be expressed in an example itself, additional constructs are provided which give hints for schema generation and validation, e.g. to express occurrence constraints. The sample implementation of Examplotron generates XSLT[17] styles from the example which can be used to validate similar documents. These XSLT styles rely on XPath for asserting that the structure of a document is equal to the structure of the example and also satisfies the additional assertions that were given with the example.

Shortcomings

Table 3.7 shows an Examplotron schema, based on the instance document from table 2.1.

This schema is very similar to the instance document and therefore easy to create and understand. However, it cannot express restrictions on the textual content of elements and attributes, and has to mix example data with additional tags to guarantee certain properties.

As Examplotron is also based on XPath, the expressive power is similar to Schematron assertions. As Schematron, Examplotron does not have the notion of a type, and therefore provides no inheritance mechanisms.

Table 3.7: Examplotron schema for scenario, step 1

```
<?xml version="1.0"?>
<!-- Examplotron schema for the scenario, step 1 -->

<pimData xmlns:eg="http://examplotron.org/0/">

  <contact eg:occurs="*">
    <Name>Peter Jones</Name>
    <Phone>0172555666</Phone>
    <Address eg:occurs="+">
      <Street>Private Drive 4</Street>
      <City>London</City>
    </Address>
  </contact>

</pimData>
```

3.8 Document Structure Description (DSD)

Document Structure Description [30] was co-developed by AT&T Labs and BRICS in November 1999 with the goals of context-dependent description of elements and attributes, flexible default insertion mechanisms and expressive power close to XSLT[17].

The possibility to define conditional constraints , e.g., depending on the content of parent attributes or elements, is one of the features that distinguishes DSD from other languages. Due to the constraint-based approach the expressiveness of the content models is remarkable, in some aspects higher than that of XML Schema. For example, a DSD schema allows to express that an attribute may only be present if another attribute is present, a very common constraint which surprisingly cannot be expressed with XML Schema. DSD also offers a powerful mechanism for default values and default content, which was inspired by Cascading Style Sheets (CSS,[4]) which are popular with HTML, but generally suited for other XML applications, too. DSD schemas can include and redefine existing schemas, but do not support an explicit inheritance mechanism. DSD does not feature a set of predefined datatypes but relies on regular expressions to flexibly specify the admissible format of strings.

Shortcomings

Table 3.8 shows a possible DSD schema for step 1 of the sample scenario.

Though DSD schemas offer a constraint-based approach, they do not provide an open-content model by default, but offer an `AnyElement` construct which "consumes" elements of any type. As in XML Schema and RELAX NG, additional efforts are therefore necessary to simulate an open-content model.

Content models in DSD are very flexible, and by offering redefinition mechanisms too flexible for our purpose: To ensure backward compatibility of a dataset the extension mechanisms for existing types must be restricted to those that are not breaking compatibility with existing datasets. The include and redefine mechanisms in DSD allow the modification of types in a way that breaks compatibility.

In general, regular expressions are more flexible than predefined datatypes. This would allow a more precise modeling of a phone number, e.g., allowing other characters such as "/" or "-" to appear, too. However, applications sharing common data in an XML document still need a set of predefined datatypes, because this set offers a basic and standardized vocabulary and syntax for the most common datatypes. This common syntax and vocabulary is important for sharing data with different applications.

Table 3.8: DSD schema for scenario, step 1

```
<?dsd URI="http://www.brics.dk/DSD/dsd.dsd"?>
<!-- scenario_step1.dsd
     DSD schema for scenario, step 1 -->

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

  <ElementDef ID="pimDataElem" Name="pimData">
    <ZeroOrMore>
      <Element IDRef="contactElem"/>
    </ZeroOrMore>
  </ElementDef>

  <ElementDef ID="contactElem" Name="contact">
    <Element Name="Name">
      <Content><StringType/></Content>
    </Element>
    <Element Name="Phone">
      <Content><StringType IDRef="PhoneNumber"/></Content>
    </Element>
    <OneOrMore>
      <Element IDRef="AddressElem"/>
    </OneOrMore>
  </ElementDef>

  <ElementDef ID="AddressElem" Name="Address">
    <Element Name="Street">
      <Content><StringType/></Content>
    </Element>
    <Element Name="City">
      <Content><StringType/></Content>
    </Element>
  </ElementDef>

  <!-- regular expression for phone numbers:
       one or more digits -->
  <StringTypeDef ID="PhoneNumber">
    <OneOrMore>
      <CharRange Start="0" End="9"/>
    </OneOrMore>
  </StringTypeDef>

</DSD>
```

Chapter 4

Requirements

This chapter lists the requirements that served as the design constraints for the XInterface schema language. Most of the requirements are generalized from the specific requirements illustrated in the sample scenario. A subsection for each requirement explains its motivation, and sometimes an additional subsection summarizes the consequences that implementing this requirement will necessarily have.

The chapter starts by introducing some definitions that are helpful in describing the requirements more precisely. Although most of the definitions are mainly used in the next chapter to model an XML document and XInterfaces, introducing the terms here enables us to express the requirements more concise.

4.1 Terminology and definitions for modeling an XML document

This document uses the model and terminology defined in the XML Information Set (XML Infoset) specification [13], which provides an abstract, syntax-free representation of the content of an XML document. As the XInterface approach does not take all information contained in the Infoset specification into account, the following model makes a few assumptions that lead to a simplified model of an XML document. This simplified model serves as the basis for describing XInterfaces and their requirements.

The simplified model ignores comment, processing instruction, document

type declaration and notation items[1] and assumes that all entity references have been expanded. Furthermore, it assumes that all namespace prefixes have been resolved to the corresponding namespace names by the namespace declarations in scope. The model therefore only considers local names and namespace names, and ignores namespace prefixes, namespace declarations and in-scope namespaces.

Definition 1 *The **fully qualified name** of an attribute or element is the combination of its local name and its namespace name. If no namespace name is available, it consists only of the local name.*

The following definition has the purpose of distinguishing the XML document that will be validated with respect to a given schema from other XML documents (e.g., documents containing the definitions of schemas to apply).

Definition 2 *The term **instance document** will be used to refer to a well-formed XML document that is subject to validation.*

As XInterfaces not necessarily define the structure of a whole XML document but will be applied to selected parts of an XML document, the following definition captures the central concept of a fragment of an XML document that can be validated against an XInterface type.

Definition 3 *An **element item** represents the information contained in a well-formed fragment of an XML document delimited by an element opening tag and its corresponding closing tag.*

In other words, an element item contains the fully qualified name of the enclosing element, all its attributes and all nested elements and the information contained therein. Note that an element item does not contain information about the context in which it occurs, such as information about parent elements. This restricts expressiveness of XInterface types to context-independent validation mechanisms.

The XML Infoset specification treats all textual (i.e., non-markup) content of an element as an ordered list of children of character items. The following definition enables us to conveniently refer to the textual content of an element.

[1] For convenience, from now on the term "notation item" and the corresponding shorthands for the other information items are used instead of the correct terms from the XML Infoset specification: "notation information item" and so on.

Definition 4 *The **character content** of an element item consists of all character items contained in its children list chunked together.*

The following instance document illustrates the definitions made so far:

Example instance document

```
<?xml version="1.0"?>
<document>
  <Address xmlns:IA="http://www.example.org/sample.xid">
    <IA:Street>Private Drive 4</IA:Street>
    <IA:City>London</IA:City>
  </Address>
</document>
```

It contains four element items (with local names `document`, `Address`, `Street`, `City`). The element item with local name `Street` has the fully qualified name `http://www.example.org/sample.xid:Street` and its character content is "`Private Drive 4`".

Element items often represent real-world concepts. In the given instance document, the element item with local name `Address` represents the concept of a postal address.

Definition 5 *An **XInterface type definition** determines a set of element items by stating requirements on them.*

At present, possible requirements are

- occurrence of an element with minimum multiplicity,
- occurrence of an attribute,
- occurrence of textual data of a certain type (either as attribute value or as character content of an element)

To validate fragments of an instance document, a schema language must select which parts of the instance document should be validated.

Definition 6 *A **context expression** selects a set of element items in a given instance document.*

In the following it is useful to distinguish elements that contain other elements from those containing only character content (or nothing).

Definition 7 *A **flat element** is an element item that does not contain nested element items.*

Definition 8 *A **compound element** is an element item that is not a flat element.*

A flat element may have character content, and in contrast to XML Schema's definition of `simpleType` a flat element may have attributes.

4.2 Open-content model

Instance documents may contain elements and attributes not specified by the XInterface type.

Rationale

An XInterface type specifies requirements for an element item, enabling the use of the specified data in an application. It has no information about other applications processing this element item and therefore about which additional information might be added. As a consequence it does not restrict the additional information in any way more than necessary.

Consequences

Restricting the cardinality of an element to an upper bound violates this requirement, as it would prevent other applications from adding more than the upper number of elements. Instead, the maximum cardinality used in our approach is interpreted as *the maximum number of elements that will be processed by that application.* A document containing more of these elements will still be valid, though a warning can be issued.

4.3 Support for multiple interfaces

The schema language should provide a way of declaring that a document implements multiple interfaces. The validation process must ensure that the requirements of all interfaces are met.

Rationale

The main purpose of the open-content model is to allow one instance document to implement multiple interfaces, offering each application its own view on the instance document.

4.4 Support for inheritance mechanism

XInterface types should be extensible, for example with an inheritance mechanism. Additionally, if the requirements of an extended interface are met, this should automatically guarantee that the requirements of the base interface are met, too. We call the property that a derived type can be used in all contexts where the base type is expected the **subsumption property**.

Rationale

Extending existing schemas promotes schema and data reuse, in particular when data evolves with the underlying concepts. For that case, the subsumption property guarantees backward compatibility of the dataset, enabling old applications to still work with the extended data. This is necessary as it cannot be assumed that all applications switch to the new data format at the same time.

4.5 Support for resolving name conflicts

The schema language should offer mechanisms to deal with name conflicts (e.g., by using XML Namespaces).

Rationale

The schema language cannot rely on local names of elements to be unique for different applications. If two applications use the same name to represent a semantic concept, one of the following two cases applies:

1. Both applications refer to the same semantic concept and therefore the data in the instance document is intended for sharing. The applications might still have different requirements for the element item in question,

and as a consequence, the conjunction of the requirements has to be fulfilled for an element item at that context.

2. The requirements refer to a different semantic concept, but unfortunately have chosen the same name to represent it. In this case the instance document must distinguish between the two different elements.

4.6 Enable easy use of instance documents in applications

It should be possible to transform a given XInterface type to a class framework that automates the process of parsing the instance document and allows easy programmatic access to the contained data that is described by the XInterface type.

Rationale

A schema does not only guarantee the format of a document (with the help of a validating parser), but contains enough information to automate the process of manually parsing an XML document (e.g., on DOM[23] or SAX[35] level) and extracting the necessary information.

An example of this approach using XML Schema is illustrated by Dashofy [14]. Interestingly, the scenario described by Dashofy also indicates problems caused by the lack of multiple inheritance in XML Schema and therefore overcoming these shortcomings was a starting point for this thesis. A similar tool that generates Java classes is also available for the RELAX Core schema language[2].

4.7 Support for typing of textual content

The schema language should provide a set of predefined simple datatypes for textual content of elements or attributes.

Rationale

Applications relying on XInterface types may not only require a specific structure of a document's markup, but also constrain the textual content of elements or attributes. For instance, a mobile phone enforces that a phone number consists of digits only.

Also, for generating a class framework as illustrated in the previous requirement, textual content must have type information to enable the mapping to constructs in the target language. For example, a flat element containing a number might be mapped to a Java member variable of type `int`.

4.8 Simplicity

The schema language should be as simple as possible.

Rationale

Feedback and criticism on XML Schema [36, 38] have shown that its complexity is a reason not to adopt it. For the requirements of the scenario it does not seem necessary to introduce complex constructs into the language. Moreover, for guaranteeing the subsumption property some constructs cannot be allowed (such as redefinition mechanisms that allow changing the type of an element or attribute) which would enhance the expressiveness of the language.

4.9 Use XML syntax

The XInterfaces schema language should use XML syntax rather than introducing another syntax.

Rationale

Using XML syntax for schema specifications enables easy processing and transforming of schemas, as there is a broad range of tools and standards available for processing XML files. It also reduces the learning curve, as most schema authors are familiar with the XML syntax.

4.10 No ordering on elements

A sequence of different elements should be treated as an unordered set, simulating a named record type. Elements with the same name need to be processed in the same order as they appear in the instance document.

Rationale

The interface approach promotes a data-centric use of XML, where the meaning of an element is in most cases independent of its position within a parent element.

Strictly demand ordering on elements leads to problems when merging multiple interfaces: If one XInterface type requires element a to appear before element b, and another requires a to appear before element c, the relative order of b and c is unspecified. A third XInterface type using b and c could decide for either of the two options. This could finally result in conflicting requirements concerning the order of elements.

The order of elements with the same name should be maintained because it may carry a semantics such as preference, priority or age. For example, an email client offers to process more than one email address, but considers the first email address found as the default one.

Consequences

As the meaning of elements in mixed content models depends on the order of text and elements, mixed content will not be supported.

4.11 Favor convenience of use instead of processing performance

The expressiveness and usability of the schema language should not be restricted because of processing performance considerations, such as requiring validation with a linear time algorithm.

Rationale

We prioritize usability and interoperability higher than processing performance. In many cases performance should not be a problem anyway, such as for the sample scenario: A contacts database with potentially a few hundred contacts should not demand a lot of processing time, even if non-linear validation algorithms are required.

4.12 Enough information for keeping validity

The instance document and the interface type definitions should contain enough information to enable changes to the dataset without losing validity of any of the required interfaces.

Rationale

If data is shared among different applications, each application might want to modify, add, or delete element items. However, the application itself can only take care of implementing the interfaces it knows about. Therefore, the instance document and the XInterface type definitions must provide sufficient information to the application to check all required interfaces. If validity of a required interface is lost because of the changes, a mechanism should be provided to automatically "repair" the dataset and restore validity.

Chapter 5

The XInterface schema language

This chapter introduces the XInterface schema language in detail. The first section introduces the basic features with a simple example. The following sections describe the syntax and semantics in detail and present the implementation of the validator and the class framework generator that was developed as part of this work.

5.1 Introduction

The XInterface language has the following constituting parts, which are described in detail in the next sections.

5.1.1 XInterface type definition

Each application using a shared instance document will define one or more XInterface types. These types should be defined in separate files, similar to the definition of a complex type in XML Schema.

The XInterface schema language is a simple schema language that provides the necessary constructs to define structure and typing requirements for element items. The language further provides means for extending existing XInterface types, allowing an explicit hierarchy of XInterface types. Syntactically, the language looks like a simplified version of XML Schema, which

should reduce the learning curve for people familiar with XML Schema. Semantically, it is similar to Schematron.

The following is a very simple XInterface type definition, requiring an element item that implements this XInterface type to have exactly one `Name` element with character content of type `string`, and at least one child element named `Address`. Each `Address` element in turn is required to have exactly one `Street` and one `City` element. The `Category` attribute on the `Address` element is optional, but if it is present, its value has to have the type `integer`. All elements and attributes have to be in the namespace `http://www.xid.org/Example.xid`[1].

Simple XInterface type definition

```
<?xml version="1.0"?>
<schema>
  <interfaceType  name="SimpleContact"
                  defaultNS="http://www.xid.org/Example.xid">
    <element  name="Name" type="string"/>
    <element  name="Address" maxOccurs="5">
        <attribute  name="Category" type="integer" minOccurs="0">
        <element    name="Street" type="string"/>
        <element    name="City" type="string"/>
    </element>
  </interfaceType>
</schema>
```

5.1.2 Implements statement

The XML document that is shared among different applications needs to be enriched with information specifying which XInterface types are to be implemented for which element items and where the XInterface type definitions can be found.

Given that the above XInterface type definition was saved in a file "example.xid", the following implements statement would require all child elements of the root element `pimData` with the local name `contact` to implement the SimpleContact interface.

[1] In this and some of the following examples a hypothetic URL "http://www.xid.org" is used as a placeholder for a location where standardized XInterface type definitions could be stored and retrieved from.

Simple Implements statement

```
<?xml version="1.0"?>

<pimData>

  <implements minOccurs="0" maxOccurs="unbounded" application="Example">
    <context>/pimData/contact</context>
    <interface name="SimpleContact" location="example.xid"/>
  </implements>

  <contact xmlns="http://www.xid.org/Example.xid">
    <Name>Peter Jones</Name>
    <Address>
      <Street>Private Drive 4</Street>
      <City>London</City>
    </Address>
  </contact>

</pimData>
```

5.1.3 Validating instance documents

Applications using XInterfaces should validate instance documents before further processing the contained information.

Figure 5.1 illustrates the validation process.

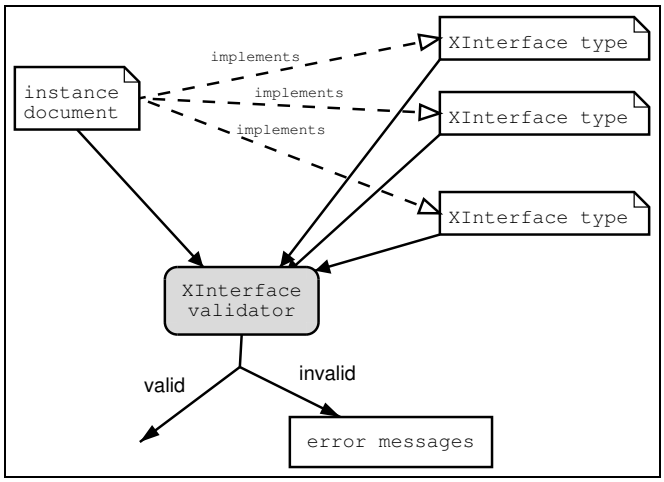


Figure 5.1: Validating an instance document

An XInterface validator takes as input an instance document and verifies that all required XInterfaces are implemented. To verify this it needs to re-

trieve and parse all XInterface types referenced in the instance document. If not all requirements are met, it should provide information on missing elements or attributes or on textual content that is incorrectly typed. Note that the validation process does *not* change the information set of the instance document. Other schema languages create normalized documents when they validate it, e.g., by inserting default values.

5.2 Syntax

The following two subsections explain the syntax of XInterface type definitions and implements statements. This syntax can be expressed also by the means of a schema language. Appendix A gives an XML Schema schema for XInterface type definitions, and an XInterface type definition for both XInterface type definitions and implements statements.

5.2.1 XInterface type definition

An XInterface type defines a content model, but does not specify which element items should be validated against that content model. Therefore an XInterface type can be checked only in connection with an implements statement that selects the element items to check.

The following example shows an XML document containing two XInterface type definitions, the first one importing the second one.

Sample XInterface types

```
<?xml version="1.0"?>
<schema>
  <interfaceType  name="MobileContact1"
                  defaultNS="http://www.xid.org/IsoContact.xid">
    <element  name="Name" type="string"/>
    <element  name="Phone" type="integer" minOccurs="0" maxOccurs="3"/>
    <element  name="Address" maxOccurs="5">
      <import   name="IsoAddress"/>
    </element>
  </interfaceType>

  <interfaceType  name="IsoAddress"
                  defaultNS="www.xid.org/IsoAddress.xid">
    <element  name="Street" type="string"/>
    <element  name="City" type="string"/>
  </interfaceType>
</schema>
```

Table 5.1: Syntax for an `interfaceType` element

Attributes	
name	*a string, required* This string uniquely identifies the interface type within the containing file.
defaultNS	*a string, optional, defaults to the empty string* This string enables to specify a default namespace for the interface type. All element and attribute declarations that do not specify a namespace inherit the value given here.
Child elements	
content declaration	*zero or more elements declaring the content model* The possible content declarations are listed in the next subsections.

A well-formed XML document can contain one or more XInterface type definitions. The XML document should have a `schema` element as its root element. XInterface types are defined as children of the root element. Depending on the situation, it may make sense to define exactly one XInterface type per XML document, or to put multiple definitions in one document. It is even allowed to place XInterface type definitions directly in an instance document. However, this does not promote the reuse of existing types and therefore is not recommended. Table 5.1 gives the syntax for an `interfaceType` element.

Content declarations

The content model defined by an XInterface type is an open-content model, i.e., it allows arbitrary additions to the instance document without causing validation errors, as long as the typing requirements of those elements that are specified are met.

The order of declarations in the content model is not significant. The required elements may appear in any order in the instance document, it is not possible to enforce a particular order. Each content declaration will be translated to an assertion that needs to be checked for the element item in question. These assertions do not allow to specify default values for elements or attributes that are not present in the instance document. Therefore XInterfaces do not change the information set of an instance document.

XInterface types do not support mixed-content models. This is a consequence of neglecting the order of elements, as in mixed-content models this order is essential for the meaning of elements. However, for most data-centric applications of XML this order is not essential for the meaning of elements and can be neglected.

The following content declarations are possible:

Flat element declaration

An example for a flat element declaration is the `Phone` element declaration of the `MobileContact1` type:

```
<element  name="Phone" type="integer" minOccurs="0" maxOccurs="3"/>
```

A flat element can carry a type definition, may carry attributes, but is not allowed to contain nested elements. Table 5.2 gives the syntax for a flat element declaration.

A flat element declaration requires that the current element item has at least `minOccurs` child elements with local name `name` and namespace name `ns`. Each of these child elements is required to have character content of type `type` and carry the correctly typed attributes specified in the attribute declarations. If `ns` is empty, all elements with local name `name` will be validated.

Compound element declaration

An example of a compound element declaration is the `Address` element declaration of the `MobileContact1` type:

```
<element  name="Address" maxOccurs="5">
  <import name="IsoAddress" location="IsoAddress.xid"/>
</element>
```

A compound element can contain subelements, but is not allowed to carry a type declaration. Table 5.3 gives the syntax for a compound element declaration.

A compound element declaration requires that the current element item has at least `minOccurs` child elements with local name `name` and namespace name `ns`. For each of these child elements, the child element's content is checked against the nested content declarations. If `ns` is empty, all elements with local name `name` will be validated.

Table 5.2: Syntax for an `element` element (flat element declaration)

Attributes	
name	*a string, required* This string defines the local name of an element in the instance document to match this requirement.
ns	*a string, optional, defaults to the defaultNS given in the* ***`interfaceType`*** *element* This string defines the namespace of an element in the instance document to match this requirement. If it is not given or empty, the value declared in `defaultNS` of the `interfaceType` element is taken.
type	*a string, optional, defaults to the empty string* This string identifies the type of the character content of the element in the instance document. The empty string represents the universal type, i.e., any type is accepted for the character content. See the next section for type systems and type identifiers that can be used.
initial	*a string, optional, defaults to the empty string* This string specifies an initial value that can be assigned to a newly created element. The initial value can be used to "repair" element items if they are created and not implementing all required interfaces. An XInterface type might use a dummy value such as "unknown" to mark element content that has not been specified yet. The initial value just ensures conformance to the interface type.
minOccurs	*a non-negative integer, optional, defaulting to 1* The semantics is the same as defined for the `implements` statement.
maxOccurs	*a non-negative integer or "unbounded", optional, defaulting to 1* The semantics is the same as defined for the `implements` statement.
Child elements	
attribute declaration	*zero or more attribute declarations*

Table 5.3: Syntax for an `element` element (compound element declaration)

Attributes	
name	*as defined above*
ns	*as defined above*
minOccurs	*as defined above*
maxOccurs	*as defined above*
Child elements	
nested content declaration	*zero or more content declarations* The content declarations nested inside this declaration will apply to the content of the enclosing declaration.

Table 5.4: Syntax for an `import` element

Attributes	
name	*a string, required* This name identifies the interface type to import.
location	*a string, optional, defaults to the empty string* If not empty, this string gives the location from where the XInterface type definition can be retrieved. It can be a relative path or an absolute URI. If empty, the file that also contains the importing XInterface's type definition is searched for the XInterface type definition to import.

Import declaration

For example, consider the `import` declaration of the `MobileContact1` type:

```
<import name="IsoAddress" location="IsoAddress.xid"/>
```

An import declaration cannot have nested content nor carry a type declaration. Table 5.4 gives the syntax for an import declaration.

An import declaration adds to the content model of the current element item the content declarations of the imported XInterface type.

Definition 9 *If an import declaration occurs at the top level of an XInterface type definition, the importing type is called* ***derived*** *from the imported type.*

An XInterface type can be derived from multiple existing types.

Attribute declaration

For example, consider the following attribute declaration (which will be later used in the `EmailContact` type):

```
<attribute  name="Category" type="string"/>
```

Table 5.5 gives the syntax for an attribute declaration.

An attribute declaration requires that the current element item carries an attribute (unless `minOccurs` equals 0) with local name `name` and namespace name `ns`. The attribute value is required to be of type `type`. The attribute declaration must not have nested elements or attributes. If `ns` is empty, all attributes with local name `name` will be validated.

5.2.2 Implements statement

The instance document that contains the interfaced data needs to be enriched with the information which interfaces are to be implemented for which elements. This information is given with the `implements` element, as shown in the following example:

Sample implements statement

```
<implements
    minOccurs   = "0"
    maxOccurs   = "unbounded"
    application = "MobilePhone1"
    comment     = "ensures a format usable for a phonebook">
  <context>/pimData/contact</context>
  <interface
      name="MobileContact1"
      location="MobileContact1.xid">
  </interface>
</implements>
```

The implements statement specifies which XInterface types will be checked and where the corresponding XInterface type definitions can be retrieved. It also allows to add information about the application that enforces this restriction on the instance document. As XInterface types will always be checked against element items, one also needs to select which element items of the instance document will be validated. A widely used specification to address parts of an XML document is XPath [12], a recommendation from

Table 5.5: Syntax for an `attribute` element

Attributes	
name	*a string, required* This string defines the name an attribute in the instance document needs to have to match this requirement.
ns	*a string, optional, defaults to the defaultNS given in the* ***`interfaceType`*** *element* This string defines the namespace of an element in the instance document to match this requirement. If it is not given or empty, the value declared in `defaultNS` of the `interfaceType` element is taken.
type	*a string, optional, defaults to the empty string* This string defines the type of the attribute content in the instance document. The empty string stands for the universal type, i.e., any type is accepted for the attribute content. See the next section for type systems and type identifiers that can be used.
initial	*a string, optional, defaults to the empty string* This string specifies an initial value that can be assigned to a newly created attribute. The initial value can be used to "repair" element items if they are created and not implementing all required interfaces. An XInterface type might use a dummy value such as "unknown" to mark element content that has not been specified yet. The initial value just ensures conformance to the interface type.
minOccurs	*0 or 1, optional, defaulting to 1* By setting this value to 0, the attribute is declared optional, otherwise it is mandatory.
maxOccurs	*fixed to 1, optional, defaulting to 1* This value is fixed to 1, as in XML there is only a single attribute of the same name allowed per element.

the W3C. By restricting the allowed XPath expressions to those that evaluate to a node-set containing only element nodes, this standard is well suited to define the set of selected element items in the `context` expression[2].

Table 5.6 gives the syntax for an implements statement.

For each implements statement, all element items selected by one of the context expressions are validated against the specified XInterface type. The occurrence constraints apply to the number of elements items that are selected by the context expressions. If there are not at least `minOccurs` element items selected, the instance document is not valid.

5.3 Semantics - informally

5.3.1 Dealing with name conflicts

The XInterface language is specifically designed to enable data sharing between applications that possibly do not know about each other. This design constraint and the feature to allow multiple inheritance of interface types inevitably leads to the problem of name conflicts, a problem that is well-known in programming languages that allow multiple inheritance (whether on interface or implementation level), and is also inherent to a meta-markup language such as XML.

To address the problem of name conflicts in XML, the W3C introduced XML Namespaces [7] in 1999, a concept to uniquely describe markup constructs with globally unique names, whose scope extends beyond the document containing them. This concept has proven itself successful and is widely adopted by the XML community. Today, the majority of the available XML tools can handle namespaces, and many XML-related specifications support it (e.g., XML Schema, XPath, XPointer, XLink).

XML Namespaces can be also utilized to deal with name conflicts in XInterfaces. Any element in the instance document may carry a namespace prefix which will be resolved to a namespace name via the corresponding namespace declaration that is in scope. In case of a name conflict, this allows elements with the same local name to be distinguished depending on their namespace.

[2] To avoid undesired side-effects of supporting the full XPath functionality, it might be useful to further restrict the set of allowed XPath expressions to a natural subset defined in XSLT[17, section 5.2] to select a node-set in an XML document. This subset basically restricts the navigation steps to the `child`, `attribute` and `descendant-or-self axis`.

Table 5.6: Syntax for an `implements` element

Attributes	
minOccurs	*a non-negative integer, optional, defaults to 1* Specifies the minimum number of occurrences that have to be found in all given contexts.
maxOccurs	*a non-negative integer or "unbounded", optional, defaults to 1* Specifies the maximum number of occurrences of that element that will be *processed* by the application implementing this interface. In contrast to XML Schema this statement does not really restrict the instance document, which can contain more nodes matching the context than in `maxOccurs` specified. This statement rather has informative character for applications and might be used by the class framework generator. Applications implementing this interface will very likely only access the first `maxOccurs` occurrences (unless `maxOccurs` set to "unbounded")and ignore all further occurrences matching the context. Note that type validation is not restricted to the first `maxOccurs` occurrences.
application	*a string, optional, defaults to the empty string* A description of the application that imposes this interface on the instance document
comment	*a string, optional, defaults to the empty string* A comment which might explain why this application enforces this interface.
Child elements	
context	*one or more elements with character content, required* One or more context elements, each specifying an XPath expressions as its character content. Any element item that is selected by one of these XPath expressions must conform to the specified XInterface type.
interface	*an element with a mandatory `name` and optional `location` attribute, required* Specifies an XInterface type to implement by `name` and gives the `location` where the XInterface type definition can be retrieved. `location` can be a relative path or an absolute URI. If it is not specified, the instance document containing the implements statement is searched for the XInterface type definition. This allows to define XInterface types within an instance document.

The attribute and element declarations in the XInterface type definition allow the specification of a namespace, too. Elements and attributes in the instance document will only be validated, if both local name and namespace name are matching. On the other hand, XInterfaces offer validation based on local name only by setting the `ns` attribute of a declaration to the empty string.

In case of a name conflict, this allows elements with the same local name to be distinguished depending on their namespace and validation with the appropriate interface type.

There is another important implication for applications that share data: As the declarations of both interface types need to specify the same combination of local name and namespace some kind of standardization mechanism is required to provide standardized fully-qualified names for elements and attributes.

One possible scenario is an organization reviewing proposals for pairs of fully-qualified names and associated semantics and publishing the accepted ones, each one documented with semantic information explaining the underlying concept. For the sample scenario we might assume an organization publishing the following list as a reference list of concepts related to addresses (the semantic information would need to be more precise and detailed in reality):

Local name	**Namespace** http://www.xid.org/	**Type**	**Semantics**
Name	IsoContact.xid	string	full name of a person
Phone	IsoContact.xid	integer	phone number, including country and area code
Address	IsoContact.xid	compound	postal address as specified in the IsoAddress namespace
...	...	...	...
Street	IsoAddress.xid	string	street part of a postal address
City	IsoAddress.xid	string	city part of a postal address

With such a list available, applications can choose to share elements with other applications by using the standardized fully-qualified names of the concepts to model, or to introduce new concepts by using their own unique namespaces. A combination of both is probably a common scenario. For example, an application is using elements from the standardized namespace `http://www.xid.org/IsoContact.xid` to model a business contact and potentially share them with other applications, but adding a `nearestSales-Office` element to it, which is qualified by a non-standardized application-

specific namespace.

While this approach is simple and clean, it depends heavily on the standardization efforts. Data sharing is only realistic if there are standardized namespaces available.

5.3.2 Merging assertions

When an XInterface type definition uses the import statement to include the content declarations of an existing type, it may happen that two content declarations have the same local name and the same namespace. These two content declarations will be merged into one assertion.

Merging two content declarations is possible, if

- both content declarations are compound element declarations
  $\Rightarrow$ The nested declarations of both will be collected and if necessary merged recursively.
- both content declarations are flat element declarations or attribute declarations and have the same type
  $\Rightarrow$ The resulting content declaration will be a flat element or attribute declaration of this type.
- both content declarations are flat element declarations or attribute declarations and one of the types is derived from the other
  $\Rightarrow$ The resulting declaration will be a flat element or attribute declaration, typed with the more restrictive type of both.

In all three cases, the new occurrence constraints will be computed as follows (the indexed variables refer to the values of the two assertions that will be merged):

$minOccurs = max(minOccurs_1, minOccurs_2)$

$maxOccurs = max(minOccurs, min(maxOccurs_1, maxOccurs_2))$

If merging is not possible, parsing of the XInterface type fails.

5.3.3 Typing of textual content

Type declarations in the XInterface language restrict the textual content of elements and attributes. A type definition can be applied to all attributes

and to all flat elements. For an instance document to be valid, the attribute value or character content of the element must satisfy the restrictions of the specified type.

The XInterface language can use any type system that allows the validation of a string against a given type. Additionally, a type hierarchy is useful that defines a derivation tree of types, enabling to check whether a type is derived from another.

Such a type system can be found in part 2 of the XML Schema specification [3]. It consists of 45 built-in simple types organized in a hierarchy[3]. As this type system is explicitly designed to be used in conjunction with other schema languages, standardized via the W3C, and a basic tool support already exists, there are strong reasons to use it. For the remainder of this work we will therefore assume that this type system will be applied.

5.3.4 Access methods

Because different applications are allowed to place different constraints on the interfaced data, modifications made by one application might cause problems threatening the validity of another application's view on the data. A typical situation might be the creation of a new element item by one applicaton where additional constraints specified by another application apply to this element but are not met. This problem is related to the problem of *View Updates* in database theory.

To ensure validity of the dataset after modifications we define the following access methods that an application modifying interfaced data must adhere to. Obviously, an application that only reads shared data is not required to follow any rules.

For simplicity the following scenario is assumed: The application reads the instance document from a file into memory, representing it there as a DOM tree. Modifications take place on this DOM representation. The application finally wants to serialize the DOM representation back to a file.

[3] As complex types go beyond placing requirements on just the text content of an element but may contain mark-up structure themselves, they cannot be used here. However, it would be an interesting starting point for a tight integration of XML Schema into XInterfaces (or vice versa) to allow any XML Schema types to be used within XInterface type definitions, and invoke an XML Schema validator for the corresponding document fragment.

Reading

For an application that modifies data we require that the validity of all interfaces must have been checked when reading, guaranteeing a valid state of the instance document before any modifications take place. As this simply means to invoke an XInterface validator when reading data this is the recommended procedure anyway.

Writing

Applications should always modify one element item corresponding to one XInterface type at a time.

Let E be the element item that an application wants to modify/create/delete, $T_1 \ldots T_i$ the XInterface types that the application itself requires to be implemented for element item E.

1. The application modifies/creates/deletes the element node in the DOM tree that corresponds to the element item E.
2. The application calls the validator to check whether this element node implements XInterface types $T_1 \ldots T_i$ (does not apply when deleting an element node).
3. The application calls the validator for all implements statements for which a member of the selected set of element items might contain the element item E.

If validation errors occur, we cant distinguish two cases and react appropriately:

1. `minOccurs` not satisfied
   (a) attribute
   $\Rightarrow$ insert the required attribute with the specified initial value
   (b) flat element
   $\Rightarrow$ insert the required element with the specified initial value as character content
   (c) compound element
   $\Rightarrow$ insert the required element with empty content

2. typing mismatch

   (a) require user to interactively modify value (e.g., with a type-specific input dialog), or

   (b) replace the invalid value with initial value

In both cases, the process needs to be repeated until the document is valid.

5.4 Semantics - formally

For evaluating the expressiveness of schema languages it is useful to model the possible classes of documents formally. A very interesting approach based on regular tree grammars can be found in Murata, Lee and Mani[37]. It classifies DTD, XML Schema, DSD, XDuce, RELAX and TREX into four classes (Regular Tree Grammars, Local Tree Grammars, Single-Type Tree Grammars and Restrained-Competition Tree Grammars) which differ in their expressiveness. Investigating XInterfaces within that formal framework still needs to be done, but might be difficult because the content model of XInterfaces is not grammar-based.

Intuitively, both instance documents and XInterface types can be modeled as a tree[4] with ordered children[5]. Determining whether a document implements an interface consists of verifying whether the instance document "contains" the tree as defined by the XInterface type as a "subtree", taking the required cardinalities into account. Figure 5.2 depicts this matching process graphically.

In the following we try to formalize the intuitive notion of "containing as a subtree", not taking a grammar-based approach but using abstract datatypes to model the class of documents defined by an XInterface type.

The notation roughly follows the standard notation used in type theory, for example by Cardelli[9]. In particular, if A and B are sets, we use

[4]Strictly speaking an XInterface type itself defines just a collection of trees, as the root element is missing. The root element is added whenever an implements statement makes use of an XInterface type. An interface type is always validated in conjunction with an implements statement, so we can simply assume this root element to be already present here.

[5]Although our model neglects the order of differently named elements, the children of an element are modeled as an ordered sequence as defined in the XML specification.

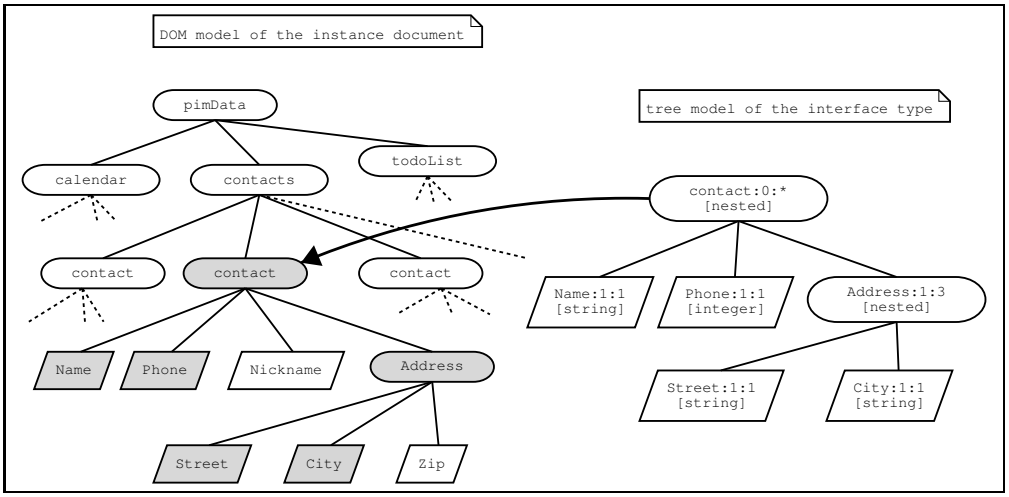


Figure 5.2: Tree model of an XInterface type

$A \times B$ to denote the Cartesian product, which is a set of all pairs with first component an element of A and second component an element of B

$A + B$ to denote the disjoint union of sets. An element of $A + B$ is either an element of A tagged with a left token (called *inl*), or an element of B tagged with a right token (called *inr*)

$(A)^B$ to denote the set of total functions mapping elements of B to elements of A

seq A to denote an ordered, possibly empty sequence of elements of A.

5.4.1 Abstract data type for an element item

XInterface types are always validated against a set of element items selected by the context expression. Thus, validating an element item E against an XInterface type T yields "valid" if the element item implements the interface type, and yields "not valid" plus some error information otherwise.

Having the definitions from section 4.1 in mind, an element item can be summarized as having a name, a set of attributes and an ordered sequence of child element items *or* character content. Thus, the content of an element

item can be modeled as

$$Contents = AttrMap \times (Text + ElemMap)$$

where

$AttrMap$	represents the set of attributes, each having a name and a value,
$Text$	represents the character content and basically can be described as a Unicode string (corresponding to PCDATA as defined in the XML specification),
$ElemMap$	represents the sequence of child element items.

If $AttrName$ represents the set of allowed attribute names, we can model the set of attributes as an element of all possible mappings from attribute names to attribute values (this allows convenient merging of multiple interfaces):

$$AttrMap = \bigcup_{Attrs \subseteq AttrName} AttrValue^{Attrs}$$

with the semantics of

for $m \in AttrMap$
$dom(m)$ is the set of attribute names for which attribute values are present

for $n \in dom(m)$
$m(n) = value$ if the attribute with name n has the value $value$.

Similarly, if $ElemName$ represents the set of allowed element names, we can represent the sequence of child element items as a relation mapping element names to a sequence of content models as follows

$$ElemMap = (seq\, Contents)^{ElemName}$$

with the semantics of

for $m \in ElemMap$
$m(n) = \langle c_1, \ldots, c_i \rangle$ if element n is exactly i times present, with the content models $c_1, \ldots, c_i$ appearing in this order.

Hence $m(n) = \langle\rangle$ if there exists no element with name n.

To exemplify the notation consider the following element item from the sample scenario:

Example element item

```
<contact        Category="private">
  <Name>Lucy Walsh</Name>
  <Phone>0179445566</Phone>
  <Address>
    <Street>Parkstr. 7</Street>
    <City>Munich</City>
  </Address>
</contact>
```

This element item could be modeled as follows:

$$c = (\{(Category \rightarrow \text{"}private\text{"})\},$$
$$inr[(Name \rightarrow (\varnothing, inl(\text{"}LucyWalsh\text{"}))), (Phone \rightarrow (\varnothing, inl(\text{"}0179445566\text{"}))),$$
$$(Address \rightarrow (\varnothing, inr[(Street \rightarrow (\varnothing, inl(\text{"}Parkstr.7\text{"}))), (City \rightarrow (\varnothing, inl(\text{"}Munich\text{"}))))]])$$

5.4.2 Interpretation of an XInterface type definition

Now that arbitrary content models can be represented with an abstract datatype, we need to specify how an XInterface type definition restricts this set to contain only the interface-valid content models.

We define an interpretation ψ that maps an XInterface type definition to a set of contents C, where each element in C is interface-valid to the XInterface type.

$\psi[]$: XInterface type definition $\rightarrow \{c \in Contents | c \text{ is interface-valid } \}$

An XInterface type definition consists of a number of assertions $a_1, \ldots, a_n$. As these assertions are independent of each other, the resulting set of valid contents is simply the intersection of all valid sets specified by each single assertion.

$$\psi[(a_1, \ldots, a_n)] = \bigcap_{1 \leq i \leq n} \psi[a_i]$$

This approach elegantly captures the semantics of importing an existing XInterface type: The set of valid content models of the existing type is intersected with the set of valid content models defined by the additional assertions. As

a side effect, this also proves the subsumption property, i.e., that any element items valid to a derived interface type are valid to the base type.

Attribute declaration

$expr =$ `<attribute name=`n `type=`t`>`

$\psi[expr] = \{(am, te) | am \in AttrMap, am(n) \in [t], te \in Text + ElemMap\}$

Compound element declaration

$expr =$ `<element name=`n `minOccurs=`m`>` $children$ `</element>`

$\psi[expr] = \{(am, em) | am \in AttrMap, em \in ElemMap,$

$em(n) = \langle c_1, \ldots, c_k \rangle, k \geq min, c_i \in \psi[children]\}$

Flat element declaration

(as children, only attribute declarations are allowed)

$expr =$ `<element name=`n `minOccurs=`m `type=`t`>` $children$ `</element>`

$\psi[expr] = \{(am, em) | am \in AttrMap, em \in ElemMap,$

$em(n) = \langle c_1, \ldots, c_k \rangle, k \geq min,$

$c_i \in \{(am, txt) | txt \in Text, txt \in [t], (am, txt) \in \psi[children]\}\}$

Import declaration

An import declaration is substituted by all declarations of the specified XInterface type. Thus, the interpretation can be applied to the inserted declarations.

5.5 Implementation

As a part of this thesis an implementation of an XInterface validator and a class framework generator were developed.

5.5.1 XInterface Validator

In general, the validation process of an XInterface validator can be outlined as follows:

for all valid implements statements in the instance document **do**
  retrieve the XInterface type from the given location and parse it
  build the set of assertions corresponding to that type
  evaluate all context expressions
  for all element items selected by the context expressions **do**
    for all assertions of the XInterface type **do**
      check the selected element item against the assertion
    end for
  end for
  count the number of selected element items
  check whether the required cardinality is fulfilled
end for

For each assertion that fails, the validator should provide enough information that an application might be able to "repair" the instance document. This information should consist of:

- an expression identifying the element item which caused the error within the document (e.g., an XPath expression)
- information about the assertion that failed

The implementation we provide is realized in Java, using the Xerces XML Parser and Xalan XPath processor, both from the Apache Project [40]. It processes both instance document and type definitions on DOM level and therefore might not be the most efficient implementation. For datatype validation it uses Sun's XML datatype library (xsdlib) [29]. Obviously, restrictions of the Xalan implementation and Sun's datatype library apply to the validator, too.

The validator implements the XInterface language as presented in this thesis and follows the validation process outlined above. To illustrate the implementation we invoke the validator on a sample instance document which contains two errors (see the comments in the input document):

```
java XInterfaceValidator document.xml
```

If the file `document.xml` contains the instance document shown in table 5.7, the validator produces the output shown in table 5.8. As expected, two error

Table 5.7: Sample input document containing errors

```
<?xml  version="1.0"?>
<!-- scenario_error.xml - sample instance document containing errors -->

<pimData      xmlns:IA="http://www.xid.org/IsoAddress.xid"
              xmlns:OC="http://www.xid.org/Organizer.xid"
              xmlns:IC="http://www.xid.org/IsoContact.xid">

  <implements minOccurs="0" maxOccurs="unbounded">
    <context>/pimData/contact</context>
    <interface name="MobileContact1" location="MobileContact1.xid"/>
  </implements>

  <contact>
    <IC:Name>Peter Jones</IC:Name>
    <IC:Phone>555666-x</IC:Phone>
        <!-- typing error: Phone does not contain an integer -->
    <OC:Phone>0171/11223344-x</OC:Phone>
        <!-- this Phone is no error: different namespace -->
    <IC:Address>
      <IA:Street>Private Drive 4</IA:Street>
          <!-- minOccurs not satisfied: missing <City> element -->
    </IC:Address>
    <OC:Email>peter.jones@web.de</OC:Email>
  </contact>

</pimData>
```

messages are given together with the the path of the element items and the assertions that caused them.

Limitations

1. This implementation of the validator cannot handle recursive types, due to the strict evaluation of import declarations. However, this should not be hard to modify in an advanced implementation.

   Specifying a recursive type in the XInterface language is possible in general by using a recursive `import` statement: Obviously, the surrounding element's `minOccurs` needs to be set to 0 to allow finite instance documents to be valid.

Table 5.8: Sample output of XInterface validator

```
user:/home/user > java XInterfaceValidator scenario.xml -d 3
XInterfaceValidator, version 0.2
Instance document (scenario_error.xml) successfully parsed
parsing implements statements...
XInterfaceType, name: MobileContact1
  [/ Name:1:1, ns: http://www.xid.org/IsoContact.xid, type: string]
  [/ Phone:0:5, ns: http://www.xid.org/IsoContact.xid, type: integer]
  [/ Address:1:5, ns: http://www.xid.org/IsoContact.xid, extendsList: IsoAddress ]
    [/Address/ Street:1:1, ns: http://www.xid.org/IsoAddress.xid, type: string]
    [/Address/ City:1:1, ns: http://www.xid.org/IsoAddress.xid, type: string]

implements statement for MobileContact1 (MobileContact1) .. valid!
node qualified with different namespace, skipping
list of errors:
* implements statement for MobileContact1 enforcing type: MobileContact1
ERROR 1:[element <Phone>: typing mismatch, expected: integer, found content:"0172/555666",
         path: /pimData/contact/IC:Phone[0], assertion:
         [/ Phone:0:5, ns: http://www.xid.org/IsoContact.xid, type: integer] ]
ERROR 2:[element <City>: minOccurs not satisfied (found: 0),
         path: /pimData/contact/IC:Address[0], assertion:
         [/Address/ City:1:1, ns: http://www.xid.org/IsoAddress.xid, type: string] ]
```

5.5.2 Class framework generator

Accessing and modifying XML documents can be done in many ways, the DOM[23] and SAX[35] models are probably the most popular ones. These standards offer very flexible approaches for parsing and modifying documents. However, for accessing typed data (whether typed with the help of DTD, XML Schema, or XInterfaces) the step of manually parsing an XML document and extracting the necessary information does not seem necessary, because the type information provides enough information to automate this process.

The XInterface language was designed with the goal of providing enough information to automatically generate a class framework for this purpose. As Java promotes the concept of interfaces, provides excellent XML support, and allows reflection on types it was chosen for mapping the XML concepts to an object-oriented programming language. As the XInterface validator already provided parts of the required functionality, the class framework generator functionality was integrated into the validator. The validator also provides the functionality of reading the content of element items into the corresponding classes. As the validator is not coupled with any of the generated classes, it uses the reflection feature of Java to create the corresponding objects and to access the `set` methods of these objects.

The following steps will be typical for developing an application that uses XInterfaces together with the class framework:

1. Design the XInterface type.
2. Call the validator to parse the XInterface type and instruct it to generate Java sources for it. The generated source code consists of one or more interfaces specifying access methods, and one or more classes representing the interfaced data contained in a matching element item.
3. Include the source codes in the Java application and use the generated interface methods to easily access the interfaced data within your application.
4. Call the validator from within your application to parse an XML document into a list of objects, each one an instance of one of the generated classes and corresponding to one valid occurrence of the specified XInterface type in the instance document.
5. The application can access the information via the access methods defined in the interfaces and change it.
6. Call the validator again for writing a list of objects back to an XML document (not implemented in this version of the class framework generator).

The class framework generator maps XInterface types to Java classes and interfaces using the following rules:

1. Attributes and flat elements that do not carry attributes are treated as **simple constructs** and will be mapped to a member in the current class. Depending on the `maxOccurs` value this member has either directly the Java type that corresponds to the specified XML Schema type (`maxOccurs` set to 1), or has the type `Vector` and will contain objects of the corresponding Java type (`maxOccurs` greater than 1).
2. Compound elements and flat elements that carry attributes are treated as **complex constructs** and will be mapped to a new class. Additionally, a member that contains one or more references to instances of that class will be added to the current class. Following the distinction from the previous case, the type of this member will either be a reference to the created class (`maxOccurs` set to 1) or a `Vector` of references (`maxOccurs` greater than 1).

3. For all members of a class the corresponding interface contains a `get`, `set` and `count` method. The `get` and `set` methods take an optional `index` argument to access specific members in case there are more than one value contained. If not given, they always access the first (and possibly only) found occurrence of the corresponding element item. The `count` method returns the number of occurrences of that element. In case of attributes and elements with `maxOccurs` set to 1, this method always returns 1.

4. For an XInterface type that imports other XInterface types, the corresponding class will be declared as implementing the Java interfaces that correspond to the imported XInterface types. The created Java interface will be declared as extending these interfaces. These declarations map the type hierarchy of XInterface types to the corresponding interface hierarchy in Java. As a consequence, programs using the class framework can make use of the subsumption property: Everywhere where the Java interface generated from an XInterface type A is expected, an interface generated from an XInterface type B can be used, if B is derived from A.

5. Each XML Schema type used for typing textual content will be mapped to a Java type. At present, of all XML Schema built-in simple types only the `string` and `integer` type are supported and mapped to the Java types `String` and `Integer`, respectively.

To understand the mapping from XInterface types to Java classes and interfaces its probably best to look at some XInterface types and their corresponding Java sources. Appendix B shows the generated source codes for some of the XInterface types used in the sample scenario. In section 6.3 a simple application using these generated sources is shown.

Limitations

The class framework generator primarily serves as a proof-of-concept and therefore has a number of limitations in its current version:

- The class framework only supports two XML Schema types (`string` and `integer`). There should be a mapping to Java types defined for all XML Schema types.

- For readability of the generated sources namespace support is not integrated. This means that the framework only takes the local names of elements and attributes into account. Name conflicts that are otherwise resolved with the help of namespaces will lead to classes with the same name.

- All generated classes are top-level classes. For a more elaborate concept, the use of Java packages could be useful, or alternatively nested classes to hold nested elements[6].

- The class framework does not support saving class content back to an XML document. Ideally, changes to element items are only done via the methods of the generated interfaces, and the class framework is called to incorporate these changes into the instance document. This would also require the implementation of the access methods from section 5.3.4, and adding methods for creating and deleting element items.

[6]This is not as trivial as it may appear, as the same type occurring on different levels of nesting will be mapped to different Java types, preventing an application from uniformly treating these types.

Chapter 6

Implementation of the scenario

We revisit the scenario from chapter 2 and show the implementation with the help of XInterfaces. First, the instance document containing the "database" is given, then all the XInterface type definitions which define the "views" for the different applications using this database are given. Finally, a sample application illustrates how the class framework allows convenient access to the data in the instance document.

For readability, the scenario (and thus, the implementation) is kept very simple. Obviously, a realistic scenario would include much more information and structure in order to be useful. Nevertheless, this scenario makes use of most of the features of the XInterface language.

6.1 Instance document

Table 6.1 shows an instance document that implements the scenario introduced in chapter 2. It contains the `implements` statements for the various applications using this data, and two `contact` elements representing a very small "contacts database".

6.2 XInterface type definitions

Each of the following XInterface type definitions define a view on the instance document ensuring that the application can make use of the data. For the namespace declarations, we assume that all namespaces beginning

Table 6.1: Instance document for scenario, step 4

```
<?xml  version="1.0"?>
<!-- scenario.xml - instance document for the scenario, step 4 -->

<pimData      xmlns:MC2="http://www.xid.org/MobileContact2.xid"
              xmlns:EC="http://www.xid.org/EmailContact.xid"
              xmlns:IA="http://www.xid.org/IsoAddress.xid"
              xmlns:IC="http://www.xid.org/IsoContact.xid">

  <implements minOccurs="0" maxOccurs="unbounded" application="MobileContact1">
    <context>/pimData/contact</context>
    <interface name="MobileContact1" location="MobileContact1.xid"/>
  </implements>

  <implements minOccurs="0" maxOccurs="unbounded" application="MobileContact2">
    <context>/pimData/contact</context>
    <interface name="MobileContact2" location="MobileContact2.xid"/>
  </implements>

  <implements minOccurs="0" maxOccurs="unbounded" application="EmailContact">
    <context>/pimData/contact</context>
    <interface name="EmailContact" location="EmailContact.xid"/>
  </implements>


  <contact  EC:Category="business" MC2:Category="1">
    <IC:Name>Peter Jones</IC:Name>
    <IC:Phone>0172555666</IC:Phone>
    <IC:Address>
      <IA:Street>Private Drive 4</IA:Street>
      <IA:City>London</IA:City>
      <MC2:Zip>7500</MC2:Zip>
    </IC:Address>
    <IC:Address>
      <IA:Street>Bahnstr. 8</IA:Street>
      <IA:City>Freiburg</IA:City>
      <MC2:Zip>79100</MC2:Zip>
    </IC:Address>
    <EC:Nickname>PJ</EC:Nickname>
    <EC:Email>peter.jones@web.de</EC:Email>
  </contact>

  <contact  EC:Category="private" MC2:Category="5">
    <IC:Name>Lucy Walsh</IC:Name>
    <IC:Phone>0179445566</IC:Phone>
    <IC:Address>
      <IA:Street>Parkstr. 7</IA:Street>
      <IA:City>Munich</IA:City>
      <MC2:Zip>80123</MC2:Zip>
    </IC:Address>
    <EC:Email>lucy.walsh@web.de</EC:Email>
    <MC2:Homepage>www.lucy-walsh.de</MC2:Homepage>
  </contact>

</pimData>
```

Table 6.2: XInterface type definition for EmailContact

```
<?xml version="1.0"?>
<!-- EmailContact.xid - interface definition for email client -->

<schema>
  <interfaceType  name="EmailContact"
                  defaultNS="http://www.xid.org/EmailContact.xid">
    <element  name="Name" type="string" initial="unknown"
              ns="http://www.xid.org/IsoContact.xid"/>
    <element  name="Nickname" type="string"
              initial="unknown" minOccurs="0"/>
    <element  name="Email" type="string"
              initial="unknown" maxOccurs="5"/>
    <attribute  name="Category" type="string" initial="unknown"/>
  </interfaceType>
</schema>
```

with `http://www.xid.org` have been standardized and their semantics represent the concepts you would expect from the given names.

Table 6.2 shows an XInterface type that could serve as a simple interface for an email client.

Table 6.3 shows the MobileContact1 type definition. This interface makes use of the import statement to import an existing Address type (given in table 6.4) into its own content model. Up to five `Phone` and `Address` elements are expected to be processed by an application using this XInterface type.

The MobileContact2 interface, given in table 6.5, extends the MobileContact1 interface with a `Homepage` element. It also extends the compound `Address` element already present in MobileContact1 with a `Zip` element.

6.3 Using the class framework

The following sample application illustrates the use of the class framework. The application makes use of the data that is specified by the MobileContact1 and MobileContact2 types of the sample scenario. The validator is invoked with the following calls to generate the necessary sources:

```
java XInterfaceValidator MobileContact2.xid -g MobileContact2
java XInterfaceValidator MobileContact1.xid -g MobileContact1
```

Table 6.3: XInterface type definition for MobileContact1

```
<?xml version="1.0"?>
<!-- MobileContact1.xid - interface definition for mobile phone -->

<schema>
  <interfaceType  name="MobileContact1"
                  defaultNS="http://www.xid.org/IsoContact.xid">
    <element name="Name" type="string" initial="unknown"/>
    <element name="Phone" type="integer" initial="0" maxOccurs="5"/>
    <element name= "Address" maxOccurs="5">
      <import name="IsoAddress" location="IsoAddress.xid"/>
    </element>
  </interfaceType>
</schema>
```

Table 6.4: XInterface type definition for IsoAddress

```
<?xml version="1.0"?>
<!-- IsoAddress.xid - standardized representation of a postal address -->

<schema>
  <interfaceType  name="IsoAddress"
                  defaultNS="http://www.xid.org/IsoAddress.xid">
    <element  name="Street" type="string" initial="unknown"/>
    <element  name="City" type="string" initial="unknown"/>
  </interfaceType>
</schema>
```

Table 6.5: XInterface type definition for MobileContact2

```
<?xml version="1.0"?>
<!-- MobileContact2.xid - interface definition for new mobile phone -->

<schema>
  <interfaceType  name="MobileContact2"
                  defaultNS="http://www.xid.org/MobileContact2.xid">
    <import name="MobileContact1" location="MobileContact1.xid"/>
    <attribute  name="Category" type="integer" initial="0"/>
    <element  name="Homepage" type="string"
              minOccurs="0" initial="unknown"/>
    <element  name="Address" maxOccurs="5"
              ns="http://www.xid.org/IsoContact.xid" >
      <element name="Zip" type="integer" initial="0"/>
    </element>
  </interfaceType>
</schema>
```

```
java XInterfaceValidator IsoAddress.xid -g IsoAddress
```

The generated sources are included in the sample application. The application itself defines a print method to access and print out the `Name`, `Phone` and `Address` elements of a Java object that implements the IMobileContact1 interface.

In the first step, the XInterface validator is called to parse the instance document and read the element items, that implement the MobileContact1 interface, into a vector of CMobileContact1 objects. As CMobileContact1 objects implement the IMobileContact1 interface, the application can call the print method for all entries of the vector.

In the second step, the application illustrates the subsumption property of derived XInterface types in Java: It reads the element items, that implement the MobileContact2 interface, into a vector of CMobileContact2 objects. As the XInterface type MobileContact2 is derived from MobileContact1, the created objects can be treated exactly as the CMobileContact1 objects, i.e., the same method for printing the content can be called. Additionally, the methods specific to CMobileContact2 can be called to access the extended data.

Table 6.6 gives the source code for the sample application. Applied to the instance document from table 6.1, it produces the output shown in table 6.7.

Table 6.6: Sample application using the class framework

```
import java.util.Vector;
/** This sample class illustrates the use of the class framework generator.
 * It uses the source codes generated for the XInterface types MobileContact1
 * and MobileContact2 to conveniently access the data contained in
 * element items implementing this interface. */
public class SampleApplication
{
  public static void main (String[] args)  throws Exception
  {
    // 1. create an instance document and parse the xml file
    InstanceDocument doc=new InstanceDocument();
    if(!doc.parse("scenario.xml"))
      return;

    // 2. collect all implements statements and build the associated types
    if(!doc.collectImplements())
      return;

    // 3. validate the MobileContact1 type
    // and read all occurrences into a list of CMobileContact1 instances
    Vector contactList=new Vector();
    if(!doc.validateOneImplements("MobileContact1", contactList))
      return;

    // 4. iterate through the list of CMobileContact1 objects
    // a) access the data with the methods of the IMobileContact1 interface
    for (int j=0 ; j<contactList.size(); j++)
      printMC1((IMobileContact1)contactList.get(j));

    // 5. validate the MobileContact2 type (which is derived from MobileContact1)
    // and read all occurrences into a list of CMobileContact2 instances
    Vector newContactList=new Vector();
    if(!doc.validateOneImplements("MobileContact2", newContactList))
      return;

    // 6. iterate through the list of CMobileContact2 objects
    // a) treat the object as instance of IMobileContact1
    // (subsumption: CMobileContact2 implements IMobileContact1)
    // b) access the extended data with the methods of the IMobileContact2 interface
    for (int j=0 ; j<newContactList.size(); j++)  {
      IMobileContact2 mc2=((IMobileContact2)newContactList.get(j));
      printMC1(mc2);                                          // a)
      System.out.println("Homepage: "+mc2.getHomepage());      // b)
    }
  }

  /** Takes an IMobileContact1 object and prints out some of the data,
   * using the get/set methods defined in the interface */
  public static void printMC1(IMobileContact1 mc1)
  {
    System.out.println("\nContact\n-------\nName: "+mc1.getName());
    for(int i=0; i<mc1.countPhone(); i++)
      System.out.println("Phone("+i+"):"+mc1.getPhone(i));
    for(int a=0; a<mc1.countAddress(); a++) {
      IAddress addr=mc1.getAddress(a);
      System.out.println("Address("+a+"):");
      System.out.println("  Street: "+addr.getStreet());
      System.out.println("  City:   "+addr.getCity());
    }
  }
}
```

Table 6.7: Output of sample application applied to instance document

```
Instance document (scenario.xml) successfully parsed

Contact
-------
Name: Peter Jones
Phone(0):172555666
Address(0):
  Street: Private Drive 4
  City:   London
Address(1):
  Street: Bahnstr. 8
  City:   Freiburg

Contact
-------
Name: Lucy Walsh
Phone(0):179445566
Address(0):
  Street: Parkstr. 7
  City:   Munich

testing subsumption property...


Contact
-------
Name: Peter Jones
Phone(0):172555666
Address(0):
  Street: Private Drive 4
  City:   London
Address(1):
  Street: Bahnstr. 8
  City:   Freiburg
Homepage: null

Contact
-------
Name: Lucy Walsh
Phone(0):179445566
Address(0):
  Street: Parkstr. 7
  City:   Munich
Homepage: www.lucy-walsh.de
```

Chapter 7

Conclusion

7.1 Summary

We have presented XInterfaces as a simple new schema language for XML. Although this language has far less powerful content models than most available schema language and lacks many of the sophisticated features of XML Schema, the taken approach still appears powerful and appealing to us. Surprisingly, the approach to define multiple views on a single XML document with the help of a schema language has not been covered by many researchers yet, and to our knowledge, none of the popular existing schema languages provide sufficient support to consequently follow this approach.

The true open-content model and the property that extended interfaces guarantee backward compatibility of the dataset are two characteristics that allow a level of interoperability and data evolution that is difficult or impossible to achieve with other existing schema languages. At the same time, the simplicity of the XInterface language makes it much more user friendly than the highly complex XML Schema.

The limited expressiveness of XInterfaces' content models certainly limits their area of application. Complex documents for one specific purpose will, without doubt, find in XML Schema a better and more powerful way to define their structure. Document-centric XML documents which rely on mixed content are also unsuited to XInterfaces. But for describing specific properties of XML documents it is easy to imagine XInterfaces coexisting with existing schema languages, and becoming a basic approach for data sharing.

The XInterface language restricts the content models to simple nesting of

elements and guarantees the subsumption property for extended interfaces. These two properties lend themselves to enable the mapping of XInterface types to an object-oriented, Java-based class framework. This class framework illustrates the amount of work that can be automated in a typical application using data described by XInterfaces, both reducing development time and guaranteeing that standard access mechanisms are used. Due to the fact that the type hierarchy of XInterfaces is mapped to a hierarchy of interfaces and classes in Java that keeps the subsumption property, deriving XInterfaces from existing ones does not only promote data reuse, but also enables code reuse. This makes applications much more stable in case of data evolution.

A lot of work needs to be done to assess XInterfaces when applied in reality. The same amount of work could potentially be spent on extending the XInterface language as presented in this thesis. The next section lists some possible issues for extending expressiveness or functionality of XInterfaces. Nevertheless, we believe that the taken approach is already valuable and usable and conclude with a vision: Section 7.3 outlines the use of XInterfaces for structuring meta-information of files, a scenario where the features of XInterfaces ideally meet the requirements.

7.2 Further work

Extend the expressiveness

At present, the XInterface language provides only support for very simple content models of element items. We propose the following improvements, but require that the current properties of XInterfaces (true open-content model, subsumption property for extended interfaces) are not lost. Therefore a careful investigation is necessary for each of the following items ensuring they can be incorporated seamlessly into the existing approach.

1. Support for a choice model, comparable to the choice pattern supported in RELAX NG.
2. Support for integrity checks with the help of IDRef attributes.
3. Support for linking mechanisms.
4. Full support for XML Schema's type system, including facets and support for user-defined types.

5. Support for mixed content models.
   Mixed content of elements could be supported by introducing pseudo-elements that contain the character content of elements. DSD also use such an approach to model the content of an element as a sequence of elements.

Other uses of XInterfaces

Apart from extending the XInterface language itself, we could also think of other usage patterns for XInterfaces.

1. Integration into XML Schema, e.g., enable the use of both XInterface types and XML Schema types in the same schema and same instance document.
2. Transform (e.g., with XSLT) an instance document with the help of a XInterface type definition to an instance document only containing the elements and attributes specified in the XInterface type.
3. Similarly, the XInterface type definition could be mapped to another schema language (e.g., XML Schema) and applied to the document generated in the previous step.
4. Validate a DOM or SAX-compliant program wrt. an XInterface type, i.e., ensure that all DOM- or SAX-operations only operate on elements and attributes specified in the type definition and do not violate typing requirements.

Extend the class framework

The class framework generator was developed as a proof of concept and therefore currently has only limited functionality. Possible improvements are:

1. Support for namespaces and therefore for dealing with name conflicts.
2. Support for writing class content back to XML documents.
   In the same way the class framework reads information from the DOM representation into the generated classes, the process of writing back the modified content of the class to the DOM representation could be automated.

3. Integrate the required access mechanisms into the class framework, enabling an application to transparently access and modify data from an instance document without being concerned with other applications using this instance document.

7.3 A vision: XInterfaces for structuring meta-information about files

XInterfaces can define and guarantee certain properties of XML files. Therefore they can describe a format for meta-information about files perfectly, as outlined in this section.

A typical scenario in the internet-based world at present is the following:
A file is sent across a network via email attachment. The person receiving the email tries to "open" the attachment (whatever that means), but the email client or file browser signals an error, because it does not recognize the file type. What is lacking is either the information of the correct file type or, if the correct file type is known, an application that can handle (e.g., view, play, edit) files of that type.

A file typically has actions associated with it. A text file may be viewed or edited, an audio file may be played. Unfortunately, in the sophisticated computer world of today, determining the actions associated with a file in many cases still seems to depend on a three-letter suffix, such as `".exe"`. Apparently, file management could profit from meta-information giving information about the content (and possibly about the associated actions) of a file.

To improve the situation described exactly above, the Multipurpose Internet Mail Extensions (MIME)[21] standard was introduced in 1992. It offers a categorization of data attached to an email according to media type and subtype information which is given in a content type header. Additionally, attributes can be attached to this header (in a `attribute=value` format) to further describe the contained data. One intention of the media type/subtype structure was the possibility of dealing with files of unknown subtype still in a meaningful way, given that the media type is known. For example, a text file with unknown subtype, but known media type "text", can still be viewed in a normal text viewer, possibly still conveying the information it is intended to carry.

While this mechanism is established for email attachments, the same meta-

information could be useful for dealing with any files (in particular multimedia files), not only those sent via email. Also, the MIME standard was invented with a strong focus on compatibility with existing email standards, and the attribute-value notation has limited expressiveness. Because of the role that XML already plays in the Internet world, an XML header seems to be a perfect fit for a successor which is not focused on email attachments but equally suited for any files. In order to become useful for file management, however, certain formats of such an "XML meta-info file" need to be standardized and adhered to. To achieve this, XInterfaces appear to be a very natural and elegant approach. Corresponding to the media types of a MIME header, generic XInterface types could specify a set of file types and the minimum information they have to carry as meta-info. Derived XInterface types could further specialize these generic types, requiring more specific information about the file.

An XInterface type "viewable" could determine that the file can be viewed on a graphics display. A subtype "viewableText" could require information about the text format of the file, enabling a file browser to call the appropriate application for viewing this particular text format. Another XInterface type "audible" could mark files that can be listened to via a speaker. If a text document is enriched with audio information, the meta-info file could simply implement both the "viewableText" and "audible" interface, allowing the user to combine both associated actions or choose the appropriate one (e.g., if a disabled person relies on audio information). Another XInterface type might be "textualDescription", which requires a textual description and/or a list of keywords describing the contents. This could provide a standard way of annotating image or audio files, and allow a textual search to be performed on all files whose meta-info file implements this interface. Very simple or even empty XInterface types could serve as a tagging mechanism, allowing one file to be in as many categories as desired.

To become successful, the standardization of these XInterface types would play a crucial role. However, this is also valid for the standardization of MIME-types today, which is the responsibility of the Internet Assigned Numbers Authority (IANA) [25].

Although only roughly investigated and outlined, it is believed that an approach combining XML meta-info files and standardized XInterface types has great potential for improving file management. Such an approach allows files to define associated actions very flexibly or with minimal effort. It also allows as many, possibly complex-structured, properties to be attached to a file as desired, and still enables a reasonable treatment of unknown or

unspecified file types as far as this is possible with the information given. Furthermore, we think that none of the popular schema languages offer the necessary constructs to implement the outlined concept of meta-information for files in such a natural and simple way.

Appendix A

Schemas for XInterfaces

Section 5.2 described the syntax of XInterface type definitions and implements statements with the help of tables listing the required attributes and elements and their meaning. We can as well describe the syntax with the help of a schema language. Table A.1 shows an XML Schema schema which defines the allowed syntax for XInterface type definitions. Table A.2 shows an XInterface type definition for the same purpose[1]. Both schemas do not distinguish between flat and compound elements, and therefore incorrectly accept typed elements with nested content.

XInterface type definitions serve a specific purpose and XML Schema therefore seems well suited for modeling their structure (possibly better than XInterfaces themselves, if an open-content model within type definitions is not desired). For modeling the structure of implements statements, however, XInterfaces are a much better fit, because they allow modeling only selected parts of an instance document. Table A.3 shows an XInterface type defining the format of an `implements` elements of an instance document. To validate the format of all implements statements in an instance document, one simply needs to add an additional implements statement, given in table A.4. This implements statement could be added to all instance documents that will be validated against XInterface types.

[1]Note that this is a recursive type definition, which at present is not supported by the provided implementation of the validator.

Table A.1: XML Schema schema for XInterface type definitions

```
<?xml version="1.0"?>
<!--  XML Schema schema for XInterface type definitions -->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!--
  <schema> must be root element of a XInterface type definition file
  each <interfaceType> child element defines one XInterface type
-->

<xsd:element name="schema">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element  name="interfaceType" type="xitd"
                    minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<!--
  an <interfaceType> element has one mandatory and one optional attribute
  and an arbitrary number of content declarations
-->
<xsd:complexType name="xitd">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="defaultNS" type="xsd:string" minOccurs="0"/>
  <xsd:sequence>
    <xsd:element ref="declaration" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>


<!--
  base element for substitution group defining a content declaration
-->
<xsd:element name="declaration" abstract="true"/>

<xsd:element  name="attribute" type="attributeType"
              substitutionGroup="declaration"/>
<xsd:element  name="element" type="elementType"
              substitutionGroup="declaration"/>
<xsd:element  name="import" type="importType"
              substitutionGroup="declaration"/>


<!-- continued on next page... -->
```

```
<!-- ...continued from previous page -->


<!-- attribute declaration -->
<xsd:complexType name="attributeType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="ns" type="xsd:string" minOccurs="0"/>
  <xsd:attribute name="type" type="xsd:string" minOccurs="0"/>
  <xsd:attribute name="initial" type="xsd:string" minOccurs="0"/>
  <xsd:attribute name="minOccurs" type="xsd:integer" minOccurs="0"/>
  <xsd:attribute name="maxOccurs" type="xsd:string" minOccurs="0"/>
</xsd:complexType>

<!-- element declaration
  we do not distinguish between flat and compound elements
  (<choice> not allowed for attributes)
  => schema incorrectly allows typed element with nested content
-->
<xsd:complexType name="elementType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="ns" type="xsd:string" minOccurs="0"/>
  <xsd:attribute name="type" type="xsd:string" minOccurs="0"/>
  <xsd:attribute name="initial" type="xsd:string" minOccurs="0"/>
  <xsd:attribute name="minOccurs" type="xsd:integer" minOccurs="0"/>
  <xsd:attribute name="maxOccurs" type="xsd:string" minOccurs="0"/>
  <xsd:sequence>
    <xsd:element ref="declaration" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<!-- import declaration -->
<xsd:complexType name="importType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="location" type="xsd:string" minOccurs="0"/>
</xsd:complexType>

</xsd:schema>
```

Table A.2: XInterface type definition for XInterface type definitions

```
<?xml version="1.0"?>
<!--
  xitd.xid - XInterface type definition for
  "XInterface type definitions"
-->
<schema>

  <implements minOccurs="1" maxOccurs="unbounded" application="xitd">
    <context>/schema/interfaceType</context>
    <interface name="xitd"/>
  </implements>

  <interfaceType name="xitd" defaultNS="">
    <attribute name="name" type="string" initial=""/>
    <attribute name="defaultNS" type="string" initial=""/>
    <import name="ContentDeclaration"/>
  </interfaceType>

  <interfaceType name="ContentDeclaration" defaultNS="">
    <element name="attribute" ns="" minOccurs="0" maxOccurs="unbounded">
      <attribute name="name" type="string"/>
      <attribute name="ns" type="string" minOccurs="0"/>
      <attribute name="type" type="string" minOccurs="0"/>
      <attribute name="initial" type="string" minOccurs="0"/>
      <attribute name="minOccurs" type="integer"  minOccurs="0"/>
      <attribute name="maxOccurs" type="string"  minOccurs="0"/>
    </element>

    <element name="element" minOccurs="0" maxOccurs="unbounded">
      <attribute name="name" type="string"/>
      <attribute name="ns" type="string" minOccurs="0"/>
      <attribute name="type" type="string" minOccurs="0"/>
      <attribute name="initial" type="string" minOccurs="0"/>
      <attribute name="minOccurs" type="integer"  minOccurs="0"/>
      <import name="ContentDeclaration"/>
    </element>

    <element name="import" minOccurs="0" maxOccurs="unbounded">
      <attribute name="name" type="string"/>
      <attribute name="location" type="string" minOccurs="0"/>
    </element>
  </interfaceType>

</schema>
```

Table A.3: XInterface type for implements statement

```
<?xml version="1.0"?>
<!--  implementsStatement.xid  -->

<schema>
  <interfaceType name="implementsStatement" defaultNS="">
    <attribute name="minOccurs" type="integer" minOccurs="0"/>
    <attribute name="maxOccurs" type="string" minOccurs="0"/>
    <attribute name="application" type="string" minOccurs="0"/>
    <attribute name="comment" type="string" minOccurs="0"/>
    <element name="context" type="string" maxOccurs="unbounded"/>
    <element name="interface">
      <attribute name="name" type="string"/>
      <attribute name="location" type="string" minOccurs="0"/>
    </element>
  </interfaceType>
</schema>
```

Table A.4: Implements statement needed for validating implements statements

```
<implements minOccurs="0" maxOccurs="unbounded"
            application="implementsChecker">
  <context>/*/implements</context>
  <interface name="implementsStatement" location="implements.xid"/>
</implements>
```

Appendix B

Source codes generated by the class framework generator

The following tables show the source codes that are generated by the class framework generator for the MobileContact1 XInterface type. To generate these sources the XInterface validator is invoked with the following calls:

```
java XInterfaceValidator MobileContact1.xid -g MobileContact1
java XInterfaceValidator IsoAddress.xid -g IsoAddress
```

The first call creates a CMobileContact1 class and a corresponding IMobile-Contact interface. It automatically creates a CAddress class and IAddress interface for the nested content of the `Address` element. As MobileContact1 imports the IsoAddress type within the `Address` element, the CAddress class implements the IIsoAddress interface and the IAddress interface extends it. Therefore the second call is necessary to create the CIsoAddress class and the IIsoAddress interface (although in this case the IAddress interface does not extend the functionality of the IIsoAddress interface, as there are no additional declarations within the `Address` element).

Table B.1: Interface source code generated for MobileContact1

```
// IMobileContact1.java (interface)
// automatically generated class framework
// generated Wed Jun 05 21:53:53 MEST 2002 by XInterfaceValidator
//
public interface IMobileContact1
{
  public String getName(int index);
  public String getName();
  public void setName(int index, String theObj);
  public void setName(String theObj);
  public int countName();

  public Integer getPhone(int index);
  public Integer getPhone();
  public void setPhone(int index, Integer theObj);
  public void setPhone(Integer theObj);
  public int countPhone();

  public CAddress getAddress(int index);
  public CAddress getAddress();
  public void setAddress(int index, CAddress theObj);
  public void setAddress(CAddress theObj);
  public int countAddress();
}
```

Table B.2: Class source code generated for MobileContact1

```
// CMobileContact1.java
// automatically generated class framework
// generated Wed Jun 05 18:49:26 MEST 2002 by XInterfaceValidator
import java.util.Vector;

public class CMobileContact1 implements IMobileContact1
{
  private String Name;
  public String getName(int index) {
    return Name;
  }
  public void setName(int index, String theObj) {
    Name=theObj;
  }
  public int countName()  {  return 1;  }
  public String getName() {  return getName(0);  }
  public void setName(String theObj) {  setName(0, theObj);  }

  private Vector Phone;
  public Integer getPhone(int index) {
    return (Integer)Phone.get(index);
  }
  public void setPhone(int index, Integer theObj) {
    if(index<0) {
      Phone.add(theObj);
    }
    else {
      Phone.add(index, theObj);
    }
  }
  public int countPhone() {  return Phone.size();  }
  public Integer getPhone() {  return getPhone(0);  }
  public void setPhone(Integer theObj) {  setPhone(0, theObj);  }

  private Vector Address;
  public CAddress getAddress(int index) {
    return (CAddress)Address.get(index);
  }
  public void setAddress(int index, CAddress theObj) {
    if(index<0) {
      Address.add(theObj);
    }
    else {
      Address.add(index, theObj);
    }
  }
  public int countAddress() {  return Address.size();  }
  public CAddress getAddress() {  return getAddress(0);  }
  public void setAddress(CAddress theObj) {  setAddress(0, theObj);  }

  public CMobileContact1()
  {
    Phone=new Vector();
    Address=new Vector();
  }
}
```

Table B.3: Interface source code generated for Address

```
// IAddress.java
// automatically generated class framework
// generated Wed Jun 05 21:53:53 MEST 2002 by XInterfaceValidator
//
public interface IAddress extends IIsoAddress
{
  public String getStreet(int index);
  public String getStreet();
  public void setStreet(int index, String theObj);
  public void setStreet(String theObj);
  public int countStreet();

  public String getCity(int index);
  public String getCity();
  public void setCity(int index, String theObj);
  public void setCity(String theObj);
  public int countCity();
}
```

Table B.4: Class source code generated for Address

```
// CAddress.java
// automatically generated class framework
// generated Wed Jun 05 21:53:53 MEST 2002 by XInterfaceValidator
//
public class CAddress implements IAddress, IIsoAddress
{
  private String Street;
  public String getStreet(int index) {
    return Street;
  }
  public void setStreet(int index, String theObj) {
    Street=theObj;
  }
  public int countStreet() {  return 1;  }
  public String getStreet() {  return getStreet(0);  }
  public void setStreet(String theObj) {  setStreet(0, theObj);  }

  private String City;
  public String getCity(int index) {
    return City;
  }
  public void setCity(int index, String theObj) {
    City=theObj;
  }
  public int countCity() {  return 1;  }
  public String getCity() {  return getCity(0);  }
  public void setCity(String theObj) {  setCity(0, theObj);  }

  // constructor
  public CAddress()  {}
}
```

Index

Bibliography

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

[2] Tomoharu Asami. *Relaxer - A Java class generator.* Internet Document, August 2000. `http://www.asahi-net.or.jp/~dp8t-asm/java/tools/Relaxer/`.

[3] Paul V. Biron and Ashok Malhotra (Eds.). *XML Schema Part 2: Datatypes.* W3C, May 2001. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.

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

[5] Ronald Bourret, John Cowan, Ingo Macherius, and Simon St. Laurent (Eds.). *Document Definition Markup Language.* W3C, January 1999. `http://www.w3.org/TR/NOTE-ddml`.

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

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

[8] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler (Eds.). *XML specification 1.0, second edition.* W3C, October 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`.

[9] Luca Cardelli. *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[10] James Clark. *Tree Regular Expressions for XML.* Internet Document, January 2001. http://www.thaiopensource.com/trex/.

[11] James Clark and Murata Makoto (Eds.). *RELAX NG specification.* Internet Document, December 2001. http://oasis-open.org/committees/relax-ng/spec.html.

[12] James Clark and Steve DeRose (Eds.). *XPath 1.0 specification.* W3C, November 2001. http://www.w3.org/TR/xpath.

[13] John Cowan and Richard Tobin (Eds.). *XML Information Set.* W3C, October 2001. http://www.w3.org/TR/2001/REC-xml-infoset-20011024.

[14] Eric M. Dashofy. *Issues in Generating Data Bindings for an XML Schema-Based Language.* Internet Document, 2001. http://www1.ics.uci.edu/~edashofy/papers/xse2001.pdf.

[15] Andrew Davidson, Matthew Fuchs, Mette Hedin, Mudita Jain, Jari Koistinen, Chris Lloyd, Murray Maloney, and Kelly Schwarzhof (Eds.). *Schema for Object-Oriented XML.* W3C, July 1999. http://www.w3.org/TR/NOTE-SOX.

[16] Dave Raggett (Ed.). *Assertion grammars.* Internet Document, May 1999. http://www.w3.org/People/Raggett/dtdgen/Docs/.

[17] James Clark (Ed.). *XSL Transformations (XSLT).* W3C, November 1999. http://www.w3.org/TR/xslt.

[18] Jon Bosak (Ed.). *XML Working Group Activity Statement.* W3C, June 1997. http://www.w3.org/XML/Activity-19970610.

[19] International Organization for Standardization. *International Standard ISO 8879 Information Processing - Text and Office Systems - Standardized Generalized Markup Language (SGML), First Edition - 1986-10-15.* October 1986. UDC 681.3.06 Ref. No. ISO 8879-1986(E).

[20] Charles Frankston and Henry S. Thompson (Eds.). *XML-Data-Reduced.* W3C, July 1998. http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm.

[21] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* The Internet Engineering Task Force, November 1996. http://www.ietf.org/rfc/rfc2045.txt.

[22] Object Management Group. *UML - Unified Modeling Language.* `http://www.omg.org/uml/`.

[23] Philippe Le Hégarel. *DOM - Document Object Model.* W3C. `http://www.w3.org/DOM/`.

[24] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Lecture Notes in Computer Science*, volume 1997, May 2000.

[25] *Internet Assigned Numbers Authority (IANA).* `http://www.iana.org`.

[26] Rick Jelliffe. *Family Tree of XML Schema Languages.* Internet Document, 1999. `http://www.ascc.net/xml/en/utf-8/family.pdf`.

[27] Rick Jelliffe. *Schematron.* Internet Document, July 2001. `http://www.ascc.net/xml/resource/schematron/schematron.html`.

[28] Kohsuke Kawaguchi. *SUN Multi-Schema XML Validator.* Juli 2001. `http://www.sun.com/software/xml/developers/multischema/`.

[29] Kohsuke Kawaguchi. *Sun XML Datatypes Library.* November 2001. `http://www.sun.com/software/xml/developers/xsdlib2/`.

[30] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. *Document Structure Description.* Internet Document, November 1999. `http://www.brics.dk/DSD/dsddoc.html`.

[31] Andrew Layman, Edward Jung, Eve Maler, Henry S. Thompson, Jean Paoli, John Tigue, Norbert H. Mikula, and Steve De Rose (Eds.). *XML-Data.* W3C, January 1998. `http://www.w3.org/TR/1998/NOTE-XML-data/`.

[32] Dongwon Lee and Wesley W. Chu. *Comparison of six schema languages.* Internet Document, June 2000. `http://www.cobase.cs.ucla.edu/tech-docs/dongwon/ucla-200008.html`.

[33] Murata Makoto. *RELAX.* Internet Document, October 2000. `http://www.xml.gr.jp/relax/`.

[34] William Matthews. *Ballmer sees XML revolution.* Internet Document, April 2002. `http://www.fcw.com/fcw/articles/2002/0415/web-xml-04-17-02.asp`.

[35] David Megginson. *SAX - Simple API for XML.* Internet Document. http://www.saxproject.org/.

[36] Anders Møller and Michael I. Schwartzbach. *XML Schema Shortcomings.* Internet Document, 2001. http://www.brics.dk/~amoeller/XML/schemas/xmlschema-problems.html.

[37] Makoto Murata, Dongwon Lee, and Murali Mani. *Taxonomy of XML Schema Languages using Formal Language Theory.* Internet Document, 2000. http://www.cs.ucla.edu/~dongwon/paper/mura0106.pdf.

[38] Jonathan Robie. *W3C XML Schema Questionaire.* Internet Document, July 2000. http://www.ibiblio.org/xql/tally.html.

[39] C. M. Sperberg-McQueen and Henry S. Thompson. *XML Schema homepage.* Internet Document, May 2001. http://www.w3.org/XML/Schema.

[40] *The Apache XML Project.* http://xml.apache.org.

[41] Henry S. Thompson. *Comment on single inheritance model.* Internet Document, June 2000. http://www.xml.com/pub/a/2000/06/xmleurope/schemas.html.

[42] Henry S. Thompson. *Email communication.* November 2001.

[43] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn (Eds.). *XML Schema Part 1: Structures.* W3C, May 2001. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.

[44] Eric van der Vlist. *Comparing XML Schema languages.* Internet Document, December 2001. http://www.xml.com/pub/a/2001/12/12/schemacompare.html.

[45] Eriv van der Vlist. *Examplotron.* Internet Document, March 2001. http://examplotron.org/.

All URLs were functional on 1.6.2002.