

Revealing the X/O impedance mismatch

(Changing lead into gold)

Ralf Lämmel and Erik Meijer

Microsoft Corp., Data Programmability Team, Redmond, USA

Abstract. We take the term X/O impedance mismatch to describe the difficulty of the OO paradigm to accommodate XML processing by means of recasting it to typed OO programming. In particular, given XML types (say, XML schemas), it is notoriously difficult to map them automatically to object types (say, object models) that (i) reasonably compare to native object types typically devised by OO developers; (ii) fully preserve the intent of the original XML types; (iii) fully support round-tripping of arbitrary, valid XML data; and (iv) provide a general and convenient programming model for XML data hosted by objects.

We reveal the X/O impedance mismatch in particular detail. That is, we survey the relevant differences between XML and objects in terms of their data models and their type systems. In this process, we systematically record and assess X-to-O mapping options. Our illustrations employ XSD (1.0) as the XML-schema language of choice and C# (1.0–3.0) as the bound of OO language expressiveness.

1 Introduction

XML is ubiquitous in today’s application development, which is otherwise biased towards the OO paradigm. XML data needs to be regularly produced or consumed by (OO) applications. Such need arises for the request/response tiers of WebServices, and the data storage tiers of many applications, as well as the import/export tiers of applications in ‘standardized’ industries. A very OO-biased developer (as opposed to an XML aficionado) expects to be able to exercise all these scenarios on the grounds of the familiar OO paradigm while taking advantage of a domain-specific object model tailored to the XML data at hand — without the need to deal with XML intimately. The so-called X/O impedance mismatch challenges this expectation.

1.1 What is the X/O impedance mismatch anyway?

Quoting Wikipedia¹, “*Impedance mismatch*’ is derived from the usage of *impedance* as a measurement of the ability of one system to efficiently accommodate the output (energy, information, etc.) of another. [...] Although the term originated in the field of electrical engineering, it has been generalized and used as a term of art in systems analysis, electronics, computer science, informatics, and physics.” In particular, the term is also used “to refer to the difficulties encountered when attempting to connect two systems which have very different conceptual bases”.

¹ http://en.wikipedia.org/wiki/Impedance_mismatch, captured on 9 January, 2007.

In the case of the X/O impedance mismatch, we are talking about the “systems” of XML and objects with their data models, programming models and type systems as conceptual bases. “Connecting” XML and objects is best called X/O mapping, also known as *XML-data binding* [10,38,31,51,11]. In particular, the X/O impedance mismatch appears to be associated with *canonical XML-to-object type-based mappings* (abbreviated as X-to-O mappings from here on). Such a mapping is meant to automatically derive object types (say, object models) from given XML types (say, XML schemas) in a canonical manner. The qualifier ‘canonical’ emphasizes that the mapping is ‘generic’ as opposed to ‘problem-specific’, (or ‘programmatic’, or ‘user-defined’).

We take the term X/O impedance mismatch to refer to the difficulty of devising a (canonical) X-to-O mapping such that the resulting, schema-derived object types (i) reasonably compare to native object types typically devised by OO developers; (ii) fully preserve the intent of the original XML types; (iii) fully support round-tripping of arbitrary, valid XML data; and (iv) provide a general and convenient programming model for XML data hosted by objects. We contend that our ‘definition’ essentially subsumes the characterizations that have been suggested elsewhere [42,58,37,62,61,17] — except that we focus on the *X-to-O* direction of mapping; we also focus on *canonical* mappings. Our hypothesis is here that the mastery of canonical X-to-O mappings is the key foundation for any other form of mechanical X/O mappings. (We will elaborate on this hypothesis.) Admittedly, ontological aspects are not covered by our definition.

1.2 An illustrative X-to-O mapping sample

Let us model simple arithmetic expressions. In XSD, we designate an element declaration `Exp` to expressions. Its content is of a complex type. We consider two expression forms (hence the `<choice>`): an int constant and a binary add expression. Thus:

```
<xs:element name="Exp">
  <xs:complexType>
    <xs:choice>
      <xs:element name="Const" type="xs:int"/>
      <xs:element name="Add">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="Exp"/>
            <xs:element ref="Exp"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

For comparison, in BNF notation:

$$\text{Exp} ::= \text{Int} \mid \text{Exp} \text{ "+" } \text{Exp}$$

For another comparison, in the notation of the Haskell programming language:

```
data Exp = Const Int | Add Exp Exp
```

X-to-O mappings may largely differ with regard to the assumed mapping rules. In this paper, we often use the `xsd.exe` tool of the .NET platform as a means to gather data points for ‘mainstream’ X-to-O mappings. For the given XML schema, we get the following C# classes:²

```
// Generated by the xsd.exe tool
```

```
public class Exp
{
    public object Item;
}

public class ExpAdd
{
    public Exp Exp;
    public Exp Exp1;
}
```

We do not attempt to explain this result here, but it is clear that these schema-derived classes do *not* compare very well to the object model that an OO developer would typically come up with. For comparison, *one* reasonable design for expression forms would be to organize the forms in an inheritance hierarchy and to label fields for subexpressions suggestively. Thus:

```
// Devised by an OO developer
```

```
public abstract class Exp { }
public class Const : Exp { public int Value; }
public class Add : Exp { public Exp Left, Right; }
```

Looking back at the automatic mapping result, one should note that we are not necessarily facing a particular defect of a specific mapping tool. (For instance, the tool can hardly come up with reasonable labels, like `Left` and `Right`, without help.) The reader is invited to try out this example with any other mapping tool, be it an implementation of the JAXB 2.0 architecture for X/O mappings (say, XML-data binding) in the Java platform [54]. Some tools may perform better than others for any specific pattern — based on subjective judgment, while also trading off simplicity, programming convenience, performance, and other factors. We contend that the impedance between the conceptual bases of XML and objects suggests that *any* mapping needs to improvise (unless AI gets involved or humans intervene).

² A note on the `xsd.exe` tool: whenever we show classes generated by the tool, we omit all generated custom attributes (‘annotations’ in Java terminology) that are meant to guide the de-/serialization framework. We also hide the ‘partial class status’ that makes the generated classes amenable to compile-time extensions. Finally, for conciseness, we use an option that prefers plain, public fields over properties (i.e., getter/setter methods) with private fields.

1.3 Dimensions of the X/O impedance mismatch

There is the *conceptual* impedance between XML and objects:

- The data models of XML and objects differ considerably, e.g.:
 - XML is based on *parented trees*; OO is based on *non-parented graphs* instead.
 - XML (but not OO) covers the concept of *mixed content* (interim text).
 - OO (but not XML) relies on *unambiguous* selectors for subcomponents.
 - XML assumes a rich *query language*; no such language is ‘native’ in OO.
- The type systems of XML and objects differ considerably, e.g.:
 - Classes aggregate members; XML leverages regular expression types instead.
 - OO-based data models use ‘flat’ classes; XML types use nested scopes.
 - XML relies on several forms of type abstractions; OO focuses on classes.
 - XML validation does not fully align with static typing for object models.

One may want to emphasize that XML and objects also differ considerably with regard to their programming models. Most notably, OO programming assumes the use of abstract data types and encapsulation. In contrast, XML processing uses ‘concrete’ XML languages (concrete data types without attached behavior) on which to devise ‘functions’ for queries and transformations. However, OO programming can also switch to the view of ‘objects as public data containers’. So we contend that the dimension of programming models is less distressing than the ones of data models and type systems. (In fact, OO programming capabilities may add value to ‘classic’ XML processing.)

X-to-O mapping is also challenged by some subordinated dimensions:

- XML is idiosyncratic, e.g.:
 - How to map XML’s element/attribute distinction to objects?
 - How to maintain XML’s processing instructions in objects?
 - How to map XML’s complicated namespaces to objects?
- The prominent XML type system, XSD, is idiosyncratic, e.g.:³
 - XSD’s counterpart for regular expression operators is non-compositional.
 - Element substitutability requires extra, auxiliary type definitions.
 - There is no general way of hiding global schema components.
- Mainstream OO languages lack desirable expressiveness, e.g.:
 - Object construction over nested types is inconvenient.
 - Non-nullable types are not yet generally available.
 - Choices (type-indexed sums) are not supported.
- XSD’s verbosity may imply that one cannot see the tree for the forest.⁴

Finally, a noteworthy dimension of the X/O impedance mismatch is the relative obscurity of the ‘X/O mapping problem’ itself. One can often find the expectation that an X/O mapping is supposed to relate an XML schema and an object model where both are *given*. When this general problem statement is chosen, then additional challenges are implied such as the potential need for a declarative mapping formalism, or the definite need for an ontological level of the mapping. For simplicity, we restrict ourselves to canonical mappings, thereby suggesting an obvious direction for future work on the X/O impedance mismatch.

³ An illustration of XSD’s idiosyncrasies: <http://www.w3.org/2002/ws/databinding/issues/37/>

⁴ An illustration of XSD’s verbosity: <http://www.charlespetzold.com/etc/CSAML.html>

1.4 The ambition: survey X/O differences

There exist many views on the X/O impedance mismatch and X-to-O mapping: Is the mismatch perhaps overrated? Should we just be fine with DOM-like XML programming? Should we add some extras to DOM? Which extras? Should we disband the so-called ‘schema first’ model, thereby making X-to-O object mapping irrelevant? And so on. We do not stake any of these views (in this paper). Instead, we dive deep into the (contrived or overrated or real) X/O impedance mismatch. *The paper’s goal is to provide the most comprehensive discussion (to date) of the differences between the data models and type systems of XML and objects. The idea is that such a survey can be used by practitioners and researchers who want to understand the mismatch and contribute to its further reconciliation in whatever way they see fit (up to the point to abandon XML entirely).* The X/O impedance mismatch has been discussed elsewhere [42,58,37,62,61,17], but the present paper is the first to systematically collect and assess the most obvious (and some of the less obvious) X-to-O mapping options.

1.5 The setup: map XSD to C#

We expect the reader to be versed in the OO paradigm and to be fluent in a statically typed OO language such as Java, Eiffel or C#. All illustrative mapping results are presented as C# fragments, but a very similar development would be possible for Java 1.5. We will mostly suffice with C# 2.0, but use the emerging C# 3.0 language [45,46] in a few places. We will explain C# specialties as we encounter them. XSD (XML Schema 1.0 [67]) is our schema language of choice. As of writing, XSD is by far the most widely used schema language, and it is very expressive (compared to DTD [68] or RELAX NG [49]). Our discussion of differences in the dimension of data models is largely independent of the choice of XSD — except that we need *some* schema notation for our samples. Our discussion of differences in the dimension of type systems focuses on issues that are likely to hold for any ‘comprehensive’ schema language.⁵ In so far that we touch upon some idiosyncrasies of XSD, the paper is somewhat specific to XSD. We do not expect the reader to be familiar with XSD, but we do expect some prior exposition to grammar-like formalisms and to XML processing of some kind (DOM, XSLT, XQuery). We will explain XSD specialties as we encounter them.

2 Background

Before we dive into the X-to-O mapping topic, let us establish additional context. In particular, let us organize possible mitigation strategies for the X/O impedance mismatch. Also, let us devise a simple programming problem that we can use for a discussion of different kinds of OO/XML programming cocktails. Ultimately, we identify desirable properties of X-to-O mappings.

⁵ In particular, we have checked that most issues would also apply to RELAX NG, even though RELAX NG is sometimes considered a simpler schema language than XSD. For instance, the mere absence of substitution constructs in RELAX NG does not imply that mapping is simplified; instead it implies that substitution relationships must be discovered on the grounds of the assumed use of ‘design patterns’ [60].

2.1 Reconciliation of the X/O impedance mismatch

The mitigation strategies are listed in no particular order.

1. *Give up on the idea of X-to-O mapping, operate on untyped, DOM-like XML trees and reserve the role of XML schemas for validation.* There are indeed proponents for this position [37]. We have argued elsewhere [33] that this position may potentially ‘throw out the baby with the bath water’. If one gives up on the idea of X-to-O mapping, as proposed, then schemas are not leveraged, in any way, to improve developer productivity by means of static typing. Also, the developer deals with XML intimately, which is not appreciated by OO-biased developers.
2. *Improve the aforementioned option by leveraging XML schemas for the design experience:* The development of VB9.0 [40] adopts this strategy. That is, schema-based IntelliSense (say, IDE support for tool tips) is under way such that it helps the programmer to guide queries over XML trees based on designated XML-member syntax and the construction of XML literals. No X-to-O mapping is performed. The developer still deals with XML intimately (just as in the case of DOM), but ‘IntelliSential’ XML types provide a design experience that compares to schema-derived object types (and the associated, conservative IntelliSense).
3. *Require an expressive OO language:* One may attempt to identify general language expressiveness (without committing to the idiosyncrasies of XML or XSD) that would simplify the X/O mapping effort. (Such expressiveness may also imply corresponding extensions of the underlying virtual machine.) For instance, the combination of generics, functional OO programming and language-integrated queries may alleviate the impedance [33]. Regular expression types (and potentially pattern matching on such types) [22] may improve static typing for schema-derived object models. For instance, the C#-like research language $C\omega$ [42,41,6] comprises an extended type system to cover essentially the EBNF-part of XSD.
4. *Create a language cocktail such that typed XML is deeply embedded into (amalgamated with) an OO language:* The development of XJ [26,25,9,28] adopts this strategy. That is, XJ amalgamates the Java and XSD type systems; XML and XPath are embedded into Java and its type system; XML trees are represented as DOM trees at runtime. Other work on embedding includes XACT [30], Xtatic [21,20] and $C\omega$. The $C\omega$ language does not opt for a full deep embedding of typed XML, but it covers the type-checked construction of XML trees on the grounds of a syntactical translation from XML literals to object construction.
5. *Devise a best-effort mapping:* one can take the position that the mismatch is overrated [61], and devise an informed and matured mapping, while accepting that some problems remain. For instance, the dreaded problem of preserving XML comments and mixed content goes away once we are willing to use DOM-like objects for the state part of the schema-derived object model [55,5,33].
6. *Profile the use of XML schemas; assuming that the profiled subset can be mapped in a satisfactory manner:* as of writing, there is a W3C Working Group for ‘XML Schema Patterns for Databinding’, which classifies XSD constructs in a way that *could* be interpreted as a profile [17] — even though the work is primarily meant to capture the degree of XSD support by actual X-to-O mapping technologies.

7. *Assume that X-to-O mapping cannot be automated in a satisfactory manner and engage in programmatic mappings:* hence an object model with de-/serialization functionality (to act as bridge between XML and objects) would be hand-crafted. The programmatic mapping may also be defined with the help of an (interactive) mapping tool. A less extreme instance of this strategy is the provision of a canonical X-to-O mapping technology with rich customization capabilities.
8. *Disband the ‘schema first’ model; proclaim the ‘code first’ model:* suitable object models would define the de-/serializable data structures for interoperable applications. XML schemas *may* be still used in describing the data interchange format, such as in the case of WCF’s data contracts for .NET types [47]. In an extreme case, XML, as such, may be disbanded for the on-wire data interchange format, such as in the case of JSON [29].
9. *Surrender and stay with a more XML-centric language:* Typically, this would imply the use of languages such as XSLT and XQuery. One may still hope to interoperate with OO applications on the grounds of an appropriate foreign-language interface or simple exchange formats.
10. *Use functional instead of OO programming:* The type systems of languages like F#, Haskell or SML with their support for algebraic data types, make it relatively easy to represent content models. Also, functional programming is generally convenient for modeling XML-like queries and transformations. For instance, these capabilities are demonstrated by existing X/O mapping technologies for Haskell [71,2]. One may still hope to interoperate with OO applications.

2.2 OO programming on XML data with DOM & Co.

X-to-O mappings deliver ‘domain-specific’ object models (for purchase orders, health-care workflows, configuration files) for OO programming on XML data. Most obviously, the domain-specific object model is meant to enable static typing, but, generally, the use of a domain-specific object model implies that the various, known capabilities of OO programming can be leveraged for XML processing. For instance, XML processing functionality can be expressed by means of (potentially virtual) methods to be attached to the schema-derived object model. Also, XML-processing code can be debugged in a type-aware manner.

Before we engage in typed XML processing based on schema-derived object models, we will illustrate ‘untyped’ XML processing, using generic, DOM-like objects. This makes it easy to pronounce the potential benefits of X-to-O mapping.

A running example

Consider the following problem:

Total price times quantity for all items of a batch of purchase orders.

```

<orders>
  <order id="47" zip="98052">
    <item id="23">
      <price>42</price>
      <quantity>2</quantity>
    </item>
    <item id="32">
      <price>33</price>
      <quantity>3</quantity>
    </item>
  </order>
  <!-- ... more orders elided ... -->
</orders>

```

Fig. 1. An XML document with a batch of purchase orders

Fig. 1 shows illustrative XML data. For the sake of a slightly more interesting example, let us assume that the total should only comprise orders with a specific zip code. Fig. 2 implements the functionality. We exercise different programming styles for the reader’s convenience.⁶

- *DOM/XPath style using the System.Xml namespace* — The relevant orders are characterized by an XPath expression that is represented as a string value and interpreted by `SelectNodes`. In particular, the XPath expression comprises a predicate to filter out the relevant orders on the basis of the `zip` attribute. The resulting node set of orders is processed in a for-each loop. Price and quantity are (laboriously) extracted from the relevant children of each order.
- *System.Xml.Linq in imperative mode* — We use LINQ to XML — a simplified XML programming API [44,39]. The child axis of XPath is replaced by the API member `Elements(...)`. The test for the zip code becomes a normal conditional in the scope of a for-each loop. The LINQ to XML API provides convenient casts to access an element’s or an attribute’s content (say, `value`).
- *System.Xml.Linq combined with LINQ syntax* — We switch to functional programming style based on LINQ’s [46] language-integrated syntax for FLWOR/SQL-like queries. That is, we aggregate the total by a query *expression*; the clauses of a LINQ expression are reminiscent of list processing in higher-order functional programming (using `map`, `filter` and `reduce`).⁷

⁶ A note on C#: throughout the paper, we tend to use `var` style variable declarations, as in `var total = ...`, as opposed to `double total = ...`, thereby relying on type inference of C# 3.0.

⁷ A note on LINQ syntax: the expression form `from x in l select y` denotes the computation of a new list l' from the given list l such that each element x of l is mapped to an element y of l' (where x is a variable and y is an expression that may refer to x). One can add `where` clauses so as to filter the list l . There are also means for grouping and ordering. One can cascade `from` clauses and nest LINQ queries.

<p><u>DOM/XPath style using the System.Xml namespace</u></p> <pre> static double GetTotalByZip(XmlDocument d, int zip) { var total = 0.0; var query = string.Format("orders/order[@zip={0}]/item", zip); foreach (XmlElement x in d.SelectNodes(query)) { var price = Double.Parse(x.SelectSingleNode("price").InnerText); var quantity = Int32.Parse(x.SelectSingleNode("quantity").InnerText); total += price * quantity; } return total ; } </pre>
<p><u>System.Xml.Linq combined in imperative mode</u></p> <pre> public static double GetTotalByZip(XElement os, int zip) { var total = 0.0; foreach (var o in os.Elements("order")) if ((int)o.Attribute("zip") == zip) foreach (var i in o.Elements("item")) total += (double)i.Element("price") * (int)i.Element("quantity"); return total ; } </pre>
<p><u>System.Xml.Linq combined with LINQ syntax</u></p> <pre> public static double GetTotalByZip(XElement os, int zip) { return (from o in os.Elements("order") from i in o.Elements("item") where (int)o.Attribute("zip") == zip select ((double)i.Element("price") * (int)i.Element("quantity"))).Sum(); } </pre>

Fig. 2. Compute the total for orders with a given zip code

DOM-like programming suffers from two shortcomings. First, the XML-processing code is not type-checked with regard to the content model for purchase orders — assuming that such a content model exists. Second, the XML-processing code deals with XML intimately; it does not resemble OO programming style — even though the problem at hand is conceptually not tied to XML. These are the main issues that X/O mapping is supposed to address.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="orders">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="order" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="order">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="item" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required"/>
      <xs:attribute name="zip" type="xs:int" use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="item">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="price" type="xs:double" />
        <xs:element name="quantity" type="xs:int" />
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

Fig. 3. An XML schema for batches of purchase orders

2.3 OO programming on XML data with ‘schema first’

We want to encode the functionality for totaling purchase orders at the level of a designated object model. Let us assume that a ‘standardization body’ has readily published an XML schema that describes the structure of purchase orders; cf. Fig. 3. Hence, the XML schema serves as the primary data model from which to derive an object model, if needed. Thus, we face a ‘schema first’ scenario as opposed to a ‘code first’ scenario.

The simple schema at hand can be mapped to an object model as follows:

- Root element declarations are mapped to classes.
- Root element names are mapped to class names.
- Local element declarations in an XSD <sequence> are mapped to fields.
- Local element names are mapped to field names.

```

public class orders
{
    public order[] order;
}
public class order
{
    public item[] item;
    public string id;
    public int zip;
}
public class item
{
    public double price;
    public int quantity;
    public string id;
}

```

Fig. 4. A schema-derived object model for purchase orders

```

public static double GetTotalByZip(orders os, int zip)
{
    return ( from o in os.order
            from i in o.item
            where o.zip == zip
            select i.price * i.quantity
            ).Sum();
}

```

Fig. 5. OO programming on XML data with ‘schema first’

- Local element types are mapped to field types.⁸

(The explored simple types `xs:string`, `xs:int` and `xs:double` have straightforward counterparts in the .NET/C# type system.) Fig. 4 shows the mapping result, when the `xsd.exe` tool is used. Fig. 5 shows the typed encoding for totaling purchase orders; see Fig. 2 again for the untyped encoding. Clearly, the typed encoding is type-checked and liberated from XML idiosyncrasies. In addition, the typed version is also considerably more concise because invocations of generic DOM-like API members are replaced by direct OO member access to objects for purchase orders.

The untyped approach of the previous subsection assumes that the XML data is held in memory based on a generic object type for XML trees. In contrast, the typed approach in the present subsection assumes that instances of schema-derived object types are

⁸ A note on XSD: We use the term *element type* to refer to the type attribute of an element declaration, if present, or to the anonymous type of an element declaration, otherwise.

```

public class orders : XElement
{
    public orders() : base("orders") { }
    public IEnumerable<order> order { get { return Elements("order").OfType<order>(); } }
}
public class order : XElement
{
    public order() : base("order") { }
    public IEnumerable<item> item { get { return Elements("item").OfType<item>(); } }
    public string id { get { return (string) Attribute("id"); } }
    public int zip { get { return (int) Attribute("zip"); } }
}
public class item : XElement
{
    public item() : base("item") { }
    public double price { get { return (double)Element("price"); } }
    public int quantity { get { return (int)Element("quantity"); } }
    public string id { get { return (string) Attribute("id"); } }
}

```

Fig. 6. XML objects for purchase orders (read only)

somehow populated with XML data. Typically, population is based on de-serialization. (In the other direction, we say that objects are serialized to XML. Elsewhere, the terms unmarshalling and marshalling are used.) As an illustration, the following code demonstrates de-serialization based on .NET's System.Xml.Serialization namespace; the code reads XML data into a new object of type `orders` before it invokes the computation of `GetTotalByZip`:

```

var serializer = new XmlSerializer(typeof(orders));
var reader     = XmlReader.Create(xmlFile);
var ords      = (orders) serializer .Deserialize(reader);
Console.WriteLine(GetTotalByZip(ords,98052));

```

2.4 Plain objects vs. XML objects

The most tempting way of thinking of schema-derived object models is indeed to anticipate ‘plain objects’ based on classes with just ‘plain fields’, as exemplified by the object model in Fig. 4. However, there is another option: the schema-derived object model may serve as an *abstract data type* for typed XML access, while hiding the particular XML-data representation. We also use the term ‘XML objects’ [33] in this case. *The XML data may be stored in a database or in an object for an untyped XML tree, while the XML objects merely define a typed view on the XML data.*

The term ‘view’ is used in a similar manner in the database domain [4,24], XPath and XSLT processing [50] or functional programming [70,12,48]. In our case, the internal state of an object is meant to account for a precise (high-fidelity) representation, whereas the interface is tailored to observations (and updates) prescribed by the schema. Throughout the paper, we will mention trade-offs regarding the options ‘plain objects vs. XML objects’; we refer to [33] for a related discussion.

Fig. 6 illustrates an instance of the notion of XML objects for our running example. That is, we use the LINQ to XML API as an ‘XML store’ while we define an object

model to provide typed access properties, which are implemented in terms of untyped API idioms.⁹ The XML object types are defined as subclasses of `XElement` — LINQ to XML’s generic type for XML trees.¹⁰ (Clearly, there are implementation options other than making `XElement` the base class of schema-derived object types.) The ‘typed’ interface provided by the XML objects is essentially the same as the one provided by the earlier plain objects — as far as typed read access is concerned. (Read access is based on ‘getters’ and the generic type `IEnumerable` for read-only lists. We omit typed setters for brevity; they are more complicated and not needed for the query in the running example.)

2.5 Object serialization based on ‘code first’

The X-to-O direction of mapping is associated with ‘schema first’. Hence, the O-to-X direction of mapping is associated with ‘code first’. In a ‘code first’ scenario, the OO developer devises an object model as the primary data model. A corresponding XML schema may be again derived in a canonical or in a problem-specific manner. The canonical option is prevalent because the routine purpose of O-to-X mappings is actually ‘object serialization’. In such a case, the derived schema serves as ‘contract’ for the ‘objects on the wire’, i.e., the serialized objects.

For the record, we should note that, in principle, object serialization, by itself, does not necessitate O-to-X mapping since an untyped XML format could suffice. In fact, object serialization does not even require the use of XML since representation formats other than XML may be used [29].

Let us illustrate ‘code first’ and a typical, canonical O-to-X mapping. We use WCF’s ‘data contracts’ [47] (which is now part of the .NET platform) as the ‘code first’ technology of choice. Several X/O mapping technologies, including `xsd.exe`, actually serve both X-to-O and O-to-X mappings, but we prefer a pure ‘code first’ technology for clarity. Fig. 7 shows an object model (for purchase orders) that is prepared for ‘data contracts’. That is, there are custom attributes (annotations in the Java terminology) to ‘mark’ the classes and the members that are meant to participate in object de-/serialization; cf. `DataContract` and `DataMember`. As it happens, the object model in the figure is identical to the earlier schema-derived object model (the ‘plain objects’ version) — except for the additional custom attributes.

⁹ A note on C#: We use the language concept of a property, i.e., data access through getters and setters. (In the figure, we only define getters.) That is, properties are OO members that facilitate abstraction and information hiding for the state part of objects. Properties may comprise a getter for read access and a setter for write access. Intuitively, properties can be used like fields. In particular, setters may appear in the position of ‘left values’, and getters in the position of ‘right values’. In a language without properties, one encodes getters and setters as methods using a name convention such as `getPrice` and `setPrice`.

¹⁰ The getter implementations invoke the generic members of the LINQ to XML API (i.e., `Element`, `Elements` and `Attribute`) and perform casts for simple-typed content in the same way as the untyped XML processing code. In the case of the compound (and repeating) subtrees, LINQ’s type-based filter method `OfType<...>` is put to work so as to ‘downcast’ LINQ to XML’s tree type `XElement` to the schema-derived object types `order` and `item`.

```

[DataContract]
public class orders
{
    [DataMember] public order[] order;
}

[DataContract]
public class order
{
    [DataMember] public string id;
    [DataMember] public int zip;
    [DataMember] public item[] item;
}

[DataContract]
public class item
{
    [DataMember] public string id;
    [DataMember] public double price;
    [DataMember] public int quantity;
}

```

Fig. 7. An object model with declarations for a canonical XML representation

The data-contracts technology indeed leverages XML for ‘objects on the wire’ including an O-to-X mapping to derive an XML schema from a given object model. The customization of the XML format is deliberately very limited: one can just customize names and order (using additional custom attributes not exercised in the example). In addition, there are some controls that serve semantical aspects: one can take precautions so as to achieve round-tripping for unknown data; one can also deal with sharing in object graphs. Fig. 8 shows the O-to-X mapping result for our running example. The derived schema differs substantially from the ‘schema first’ version.¹¹ Clearly, the derived XML schema is more complex than the handcrafted one, but this is of no real concern — *if* we assume that the XML schema is not to be used directly by any developers. In particular, the XML representation of ‘objects on the wire’ is not to be the target of any ‘native’ XML processing code. In the case of data contracts, the serialization technology is indeed the only authority that needs to deal with XML directly.

¹¹ For completeness’ sake, here is a list of differences:

- All content is mapped to elements; attributes are not used.
- A designated XML namespace for data contracts is used.
- At a given level, particles are sorted alphabetically.
- Arrays are mapped to wrapper elements with repeating elements inside.
- Most particles are said to be nillable, i.e., their content may be omitted.
- Named complex types are used (as opposed to anonymous ones).

```

<xs:schema xmlns:tns="http://schemas.datacontract.org/2004/07/"
  elementFormDefault="qualified"
  targetNamespace="http://schemas.datacontract.org/2004/07/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="orders">
    <xs:sequence>
      <xs:element minOccurs="0" name="order" nillable="true"
        type="tns:ArrayOforder" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="orders" nillable="true" type="tns:orders" />

  <xs:complexType name="ArrayOforder">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded"
        name="order" nillable="true" type="tns:order" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="ArrayOforder" nillable="true" type="tns:ArrayOforder" />

  <xs:complexType name="order">
    <xs:sequence>
      <xs:element minOccurs="0" name="id" nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="item" nillable="true" type="tns:ArrayOfitem" />
      <xs:element minOccurs="0" name="zip" type="xs:int" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="order" nillable="true" type="tns:order" />

  <xs:complexType name="ArrayOfitem">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded"
        name="item" nillable="true" type="tns:item" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="ArrayOfitem" nillable="true" type="tns:ArrayOfitem" />

  <xs:complexType name="item">
    <xs:sequence>
      <xs:element minOccurs="0" name="id" nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="price" type="xs:double" />
      <xs:element minOccurs="0" name="quantity" type="xs:int" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="item" nillable="true" type="tns:item" />

</xs:schema>

```

Fig. 8. A derived XML schema for purchase orders

In general, it may be that the derived XML schema does not just serve as a ‘contract’ and that XML does not just serve as a representation format. Instead, the XML format may be subject to ‘native’ XML processing, in which case the derived schema should be of help for the typed XML programmer. For instance, it is not uncommon to encounter the expectation that data modeling with XSD can be circumvented by means of object modeling (and O-to-X mapping). In such a case, O-to-X mappings would be

expected to deliver schemas that reasonably compare to native schemas typically devised by an XML & XSD expert. We contend that the biggest challenge, in this case, is the conceptual clash between object *graphs* vs. XML *trees*; cf. Sec. 3.1. We either need to restrict the use of the object model to trees or serialize arbitrary object graphs as trees with the encoding of object identities or ownerships, which implies complications for the XML programmer. The ‘trees vs. graphs’ problem also shows up in the X-to-O direction. In the sequel, we focus on X-to-O mappings, indeed.

2.6 Properties of X-to-O mappings

Before we start to list X-to-O mapping options, let us identify important properties of X-to-O mappings, such as their completeness with regard to the supported schema constructs or the palatability of the derived object models. While all the properties to come are easily explained and intuitively desirable, the full set is hard (or impossible) to deliver consistently by X-to-O mappings (because of the X/O impedance mismatch).

Acceptor completeness

A mapping should accept any valid XML schema. Also, the resulting object model should accept any XML document that is valid with regard to its associated schema. (That is, de-serialization must not throw.) Giving up on mapping completeness may be acceptable, if the supported ‘schema profile’ is well defined. In contrast, we definitely face a correctness issue in the case of an object model that ‘rejects’ a valid XML instance (during de-serialization).

Schema-constraint preservation

Ideally, every schema constraint would have a counterpart in the derived object model. We need to be prepared for two forms of counterparts. First, some constraints may be mapped to object-modeling concepts whose *static typing* faithfully corresponds to schema validation. Second, other constraints may be mapped to object-modeling concepts whose *dynamic semantics* faithfully corresponds to schema validation or constraint enforcement.

While acceptor completeness is ‘state of the art’ (modulo bugs), this is not the case for schema-constraint preservation. Admittedly, full preservation of schema constraints is very challenging. For instance, the `xsd.exe` tool does not preserve any of the following schema constraints:

- The element types of the branches in a choice; cf. Sec. 4.2.
- Mandatoriness (as opposed to optionality); cf. Sec. 4.1.
- Simple- and complex-content restrictions; cf. Sec. 4.8.
- The various identity constraints.

Palatability

This term has been coined in [17] and it makes nicely clear that we are entering an informal dimension here. Ideally, when mapping a given schema to an object model, the result should reasonably compare to a ‘native object model’, i.e., an object model that an OO developer would typically devise. This requirement is hard to measure, but our introductory example illustrated violations of palatability. For instance, a content model with recurrent element names led to name mangling (recall the field names E_{xp} and E_{xp1}). We will encounter many cases of unpalatability in the subsequent sections.

Programmability

Related to palatability is the concept of programmability. That is, we expect the schema-derived object model to be convenient in programming on XML data, when compared to native forms of XML processing such as XSLT or XQuery, while assuming equal familiarity with OO and XML programming. In particular, we expect to carry out queries, updates and construction in a convenient manner.

Programmability is a property that sets X-to-O mappings apart from the neighboring field of *schema-aware XML-to-relational mappings* [8,7,1,52]. An X-to-R mapping mainly focuses on the definition of a storage representation for XML data in a database such that the XML data can be efficiently retrieved and updated. It is not necessary to accommodate a rich XML programming model *directly* on the shredded XML data. In contrast, an X-to-O mapping should deliver object models whose interface is readily convenient for devising data processing functionality on the XML data.

XML fidelity

We use the term ‘XML fidelity’ to refer to the capability of a schema-derived object model to grant access to all facets of XML data by observations (O), to maintain them along transformations (T), and to construct them (C). Here is a list of facets and an illustrative compliance matrix for `xsd.exe`:

Facet of XML data	O	T/C
Structured content (flat composites)	+	+
Structured content (nested composites)	+	?
Mixed content (interim text)	?	?
Whitespace	–	–
XML comments	–	–
Processing instructions (PIs)	–	–
Order of attributes	–	–
Namespace prefixes	–	–
Embedded DTDs	–	–
Encoding (L¨mmel vs. L¨mmel)	–	–

Legend:

– “+” — Fidelity definitely holds.

```

public static T InOutSequence<T>(string filein, string fileout) {
    var reader = new StreamReader(filein);
    var serial = new XmlSerializer(typeof(T));
    var t      = (T)serial.Deserialize(reader);
    reader.Close();
    var writer = new StreamWriter(fileout);
    serial.Serialize(writer, t);
    return t;
}

```

Fig. 9. Serialization + de-serialization sequence

- “-” — Fidelity does not hold.
- “?” — Sloppy fidelity.
- The bars in the table separate groups of aspects that can be best labeled as follows:
 - Structured content.
 - Important infocet-like data [66].
 - Lower-level representation facets of XML data.

Round-tripping

The property of maintaining facets of XML data along transformations can be formalized as a more specific round-tripping property. That is, a de-serialization phase (or a ‘load’ operation) followed by an immediate serialization phase (or a ‘save’ operation) is supposed to preserve all (or some) facets of XML data.

As a clarification, Fig. 9 shows a C# function that would be helpful in establishing round-tripping compliance or revealing violations thereof. The function exercises de-serialization, followed by serialization for a given type, input file and output file based on the System.Xml.Serialization namespace that goes with the `xsd.exe` tool. We can then use (external) ‘diffing’ functionality to compare the original input and the result of the de-serialization/serialization sequence.

Fig. 10 runs an XML schema by `xsd.exe`; it turns out that the order of structured content is not preserved by the plain de-serialization+serialization sequence. The particular example exercises nested composites, and round-tripping fails because the schema-derived object type does not maintain nesting (neither at the level of static type structure, nor by means of extra ‘housekeeping state’).

Extensibility and customizability

It is not uncommon to hear of a requirement that the mapping result must be (in some sense) extensible or customizable. In fact, the borderline between mapping customization and ‘programmatically mapping’ remains blurred until we find a reasonable definition of both terms. For instance, extra methods or state may need to be accommodated by the resulting object model, e.g., for the purpose of event handling. Compile-time class

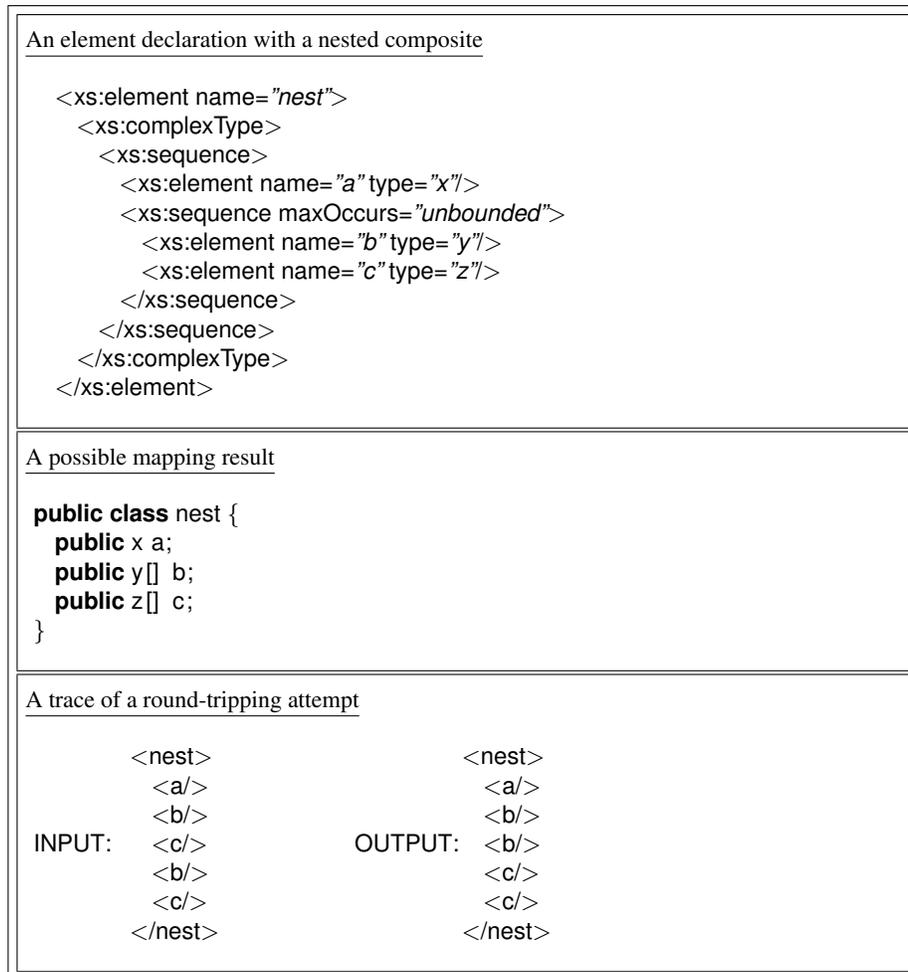


Fig. 10. Illustration of a round-tripping violation

extension (such as with partial classes of .NET) or post-compile-time class extension (such as with extension methods of the C# 3.0 language [45,46]) are specific linguistic means of tackling such extensibility and customizability. Also, the mapping itself may be subjected to a requirement for customization. For instance, it may be desirable to turn XML-data binding off for some parts of the schema so that the corresponding XML data is exposed through a generic, DOM-like API as opposed to schema-derived object types. For the record, XML objects (cf. Sec. 2.4) make this particular kind of mapping customization unnecessary since both typed and untyped access would be provided anyhow.

3 The X/O data models

The crux of the X/O impedance mismatch lies in the difference between the data models of X and O, say their ‘semantic domains’ with the *essential operations on XML data and objects*.¹² In the present section, we characterize these domains and point out differences that may contribute to the X/O impedance mismatch. To give a simple example, XML distinguishes element vs. attributes, whereas objects are composed from sub-objects in a uniform manner, and hence XML’s distinction may be hard to represent by objects. The following subsections contrast ‘X vs. O’ in the following respects:

1. Trees vs. graphs
2. Accessible vs. unavailable parents
3. Ambiguous vs. unique nominal selectors
4. Queriable trees vs. dottable objects
5. Node labels vs. edge labels
6. Ordered vs. unordered edges
7. Qualified vs. local selectors
8. Semi-structured vs. structured content
9. Tree literals vs. object initialization

3.1 Trees vs. graphs

The semantic domain for objects is essentially a certain class of (constructable, navigable, updatable) graphs. In contrast, the semantic domain for XML is essentially a certain class of (constructable, navigable, queriable, and potentially updatable) trees. When object graphs are used to represent XML trees, then the object semantics may give *too much freedom* to the programmer who is operating on XML data in objects. That is, one can freely create object graphs with sharing or cycles.

When a violation of the intended tree invariant of XML data goes unnoticed, then this may lead to problematic serialization behavior. Different things may happen: serialization may loop, run out of resources, or throw due to cycle detection — depending on the technology in question. Fig. 11 illustrates this issue. The object model at the top admits the construction of cyclic object graphs. The shown XML schema in the middle of the figure was derived from the object model (using the `xsd.exe` tool in the O-to-X direction). Of course, the element declaration is recursive, but without the admitted semantics of data-level recursion. When we execute the test code, given at the bottom of the figure, serialization throws due to cycle detection.

Hence, the use of object types imposes an obligation on the programmer: ‘Do not create cycles!’ For sharing (without cycles), the situation is perhaps less severe; depending on the technology in question, sharing may be admitted modulo the potentially too

¹² For the record, the official definition of the semantic domain for XML is the XML information set (‘infoset’ [66]), which covers *observation operations* on XML trees. (Observations may be rendered as queries in a language like XPath [65]). XML support for imperative OO languages also provides *update operations*, such as those defined by DOM [64]. Ultimately, *validation* is a further assumed operation (a predicate, in fact) on XML trees.

<p>An object model that admits cycles</p> <pre> public class Foo { public Foo bar; } </pre>
<p>A derived schema for the above object model</p> <pre> <xs:element name="Foo" nillable="true" type="Foo"/> <xs:complexType name="Foo"> <xs:sequence> <xs:element minOccurs="0" name="bar" type="Foo"/> </xs:sequence> </xs:complexType> </pre>
<p>An illustrative use of the object model</p> <pre> // Create cyclic object var x = new Foo(); x.bar = x; // Serialize to XML var myWriter = new StreamWriter("foo.xml"); XmlSerializer serializer = new XmlSerializer(typeof(Foo)); serializer .Serialize (myWriter,x); // THROWS! </pre>

Fig. 11. Illustration of a serialization problem for object graphs

implicit effect of losing sharing along serialization. (As a result, object graphs with sharing do not round-trip.) The typical OO type system is not helpful in enforcing the tree invariant. One can imagine idiomatically implemented object types (on the grounds of ‘XML objects’; cf. Sec. 2.4) that enforce the tree invariant at runtime, through dynamic checks based on extra ‘housekeeping state’, potentially complemented by a cloning semantics. Such a tree-like semantics for objects is not straightforward and may be considered unpalatable.

In principle, one can serialize object graphs as XML trees (by using generated ids in the serialized XML). However, such an approach is not helpful in the X-to-O direction of mapping, i.e., for the ‘schema first’ model, unless the given XML schemas readily used ids/idsrefs or other means of representing graph shape.

3.2 Accessible vs. unavailable parents

XML’s semantic domain does not just assume the tree invariant for XML data, it actually assumes that *tree shape is observable* in the sense that one can navigate from a subtree to its parent. The semantic domain of objects does not provide any counterpart. As a result, certain idioms of XML processing are no longer expressible when

```

public static XElement ItemsPerZip(IEnumerable<XElement> items)
{
    return
        new XElement("groups",
            from i in items
            group i by i.Parent.Attribute("zip").Value into g
            select
                new XElement("group",
                    new XAttribute("zip", g.Key),
                    g.Elements());
}

```

Fig. 12. Untyped LINQ to XML code for grouping items by zip code

```

public static bool EditAddress(XElement addr)
{
    XElement edit = new XElement(addr); // Clone for editing
    bool change = ModalXmlEdit(edit); // GUI data binding
    if (change) addr.ReplaceWith(edit); // Replace if necessary
    return change;
}

```

Fig. 13. In-place data manipulation for GUI data binding of XML tree (an address)

XML data resides in ‘plain objects’. One can imagine idiomatically implemented object types (on the grounds of ‘XML objects’; cf. Sec. 2.4) that provide parent access.

Consider the following illustrative assignment:

Suppose we want to process a list of order items (as opposed to a batch of orders). The idea is that this list may be pre-selected by a different program component. For instance, this list may comprise all those items (from a batch of orders) that require a backing purchase before sale can be confirmed. Further suppose that the processing functionality must determine the zip code per item, be it to group the items per zip code so that the transport for the backing purchases can be organized per zip code.

In this case, we need access to the parent axis so that we can retrieve the zip code for each item. Fig. 12 shows a corresponding function that groups the incoming list of items by zip code.¹³

The expressiveness of the parent axis is not limited to queries, but it also applies to updates. For instance, due to the parent axis, a reference to an XML (sub)tree is

¹³ A note on LINQ constructs: we use the LINQ notation for grouping items in a collection by a grouping key; cf. “**group ... by ... into ...**”. The grouping operation returns a nested collection where each outer item (a group) is paired with a grouping key.

sufficient to replace the tree. (Clearly, a plain object reference cannot facilitate such in-place updates.)

Fig. 13 shows a function `EditAddress` with an argument for an address (an XML tree) to be edited. The actual editing phase operates on a cloned XML tree, thereby enabling a simple means of an UNDO capability. Per convention, the call of the function `ModalXmlEdit` returns `true`, if the editorial changes are to be committed back to the caller of `EditAddress`. Hence, if `true` is returned, then the original XML tree is replaced by the changed clone. To this end, we use the `ReplaceWith` operation, which is one of the update operations of the LINQ to XML API. Clearly, this update operation must navigate to the parent and update its child list.

3.3 Ambiguous vs. unique nominal selectors

Given an object, its sub-objects are uniquely addressable by means of ‘edge labels’, i.e., field or property names. No such unique, *nominal* selectors are generally available for XML trees. There are unique *positions* for each subtree, but name-based selection would be based on ‘node labels’, i.e., element names, and these can be ambiguous.

An element declaration may carry a non-default `maxOccurs` constraint, in which case contiguous subsequences of subtrees in a tree may carry the same label. More severely, a content model may involve multiple element declarations with the same name. As a result, an otherwise tempting 1:1 mapping from element particles (of a given content model) to field or property declarations (of a corresponding object type) is problematic. For instance, how do we map the following content model?

```
<xs:element name="line">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="point" type="pointType"/>
      <xs:element name="point" type="pointType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

(We saw a similar example in the introduction of the paper.) In mainstream OO languages, we cannot have multiple fields with the same name, `point`. A typical workaround, chosen by X-to-O mapping tools, is to engage in ‘name mangling’. For instance, the two different occurrences of the element name `point` would result in two field or property names `point` and `point2`.

Instead of name mangling we may (or perhaps we should) adopt a different view on the problem. That is, we could give up on the 1:1 mapping from element particles to OO members. Instead, the mapping would be 1:1 from element *names* to OO members. An OO member may be of a collection type so as to reflect that multiple element declarations are covered (in addition to the trivial case of a single declaration for a repeating element). In the example, the resulting object type provides a single field `point` of a list type; cf. Fig. 14. Read access (say, ‘query mode’) for such OO members may be seen as an XPath-like child axis. Write access (say, ‘update mode’) is potentially challenging for content models with nested composites; cf. Sec. 4.3.

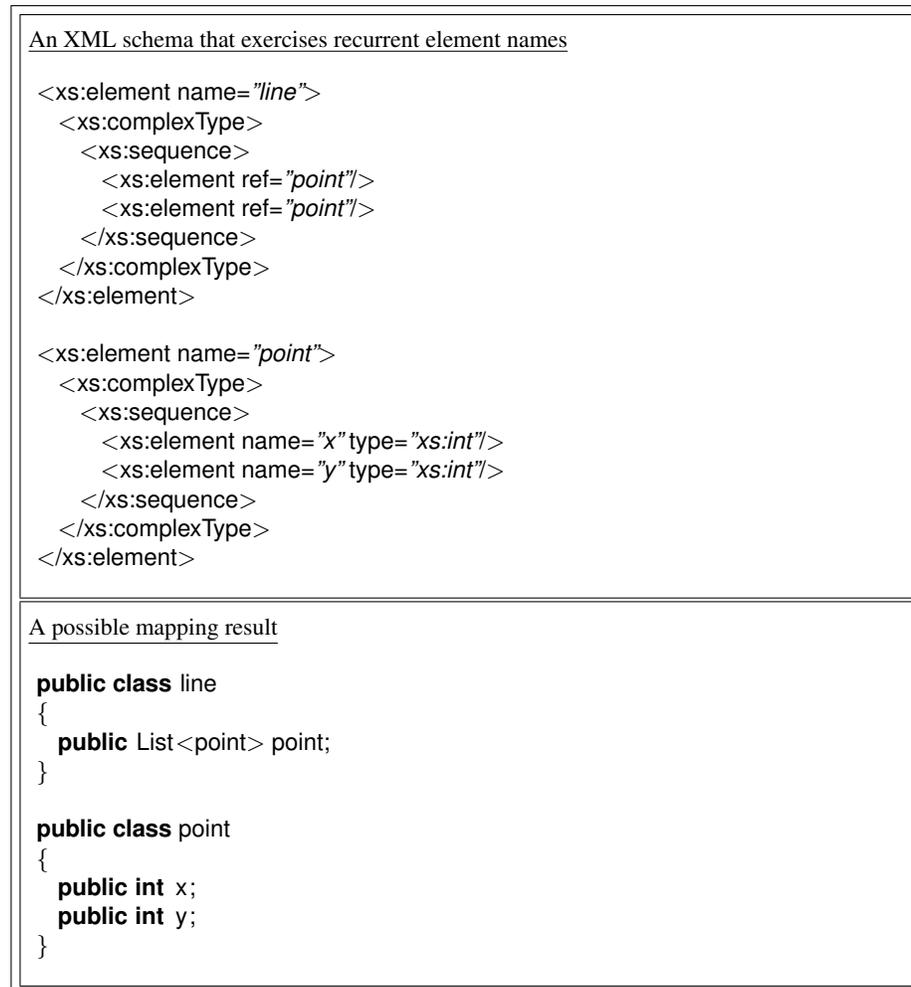


Fig. 14. Illustration of a mapping rule for recurrent element names

This (uncommon) approach is the required basis for a correct mapping such that XPath queries (using the child axis) can be mapped to object queries in a semantics-preserving manner. That is, for a given XML tree t and an XPath query q , we should be able to obtain the same objects in the following two ways: (i) evaluate q on t and map the result to objects; (ii) map q to an object query q' , map t to an object on which to evaluate q' .

Instead of using a more involved mapping rule, we may also require a slightly more powerful OO language. The C#-like research language $C\omega$ [42,41,6] provides a type language that admits recurrent field names. $C\omega$ generalizes normal member access (“.”) so that the multiple occurrences can be referred to by the recurrent name. The update direction for generalized member access is a subject for future work.

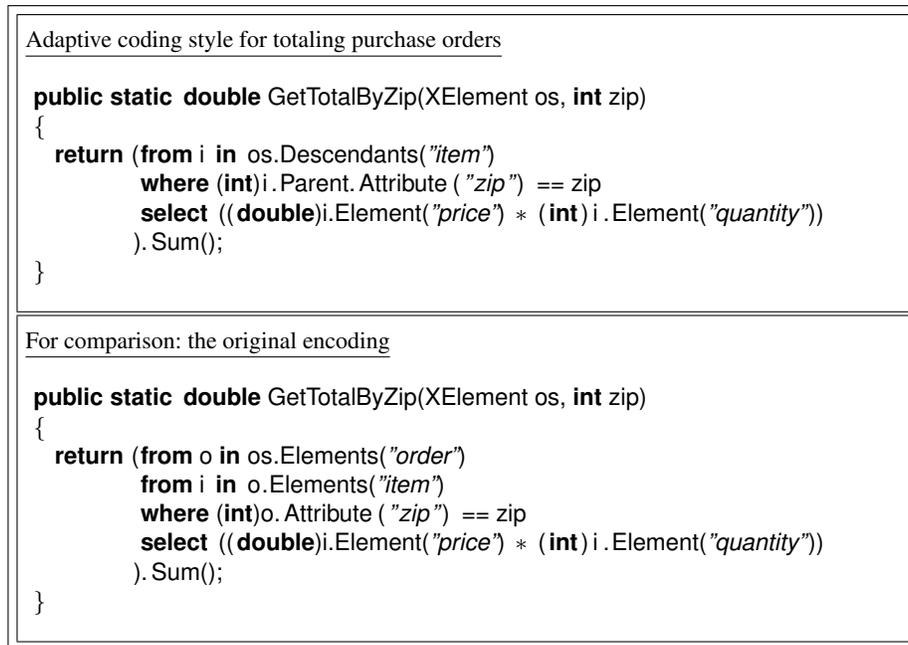


Fig. 15. Illustration of the use of the descendant and parent axes

3.4 Queriable trees vs. dottable objects

We have already mentioned the additional parent axis for XML trees. Also, we have explained that an XPath-like child axis is different from the mainstream member access (“.”) for objects. There are further axes for XML queries that do not come with any obvious counterpart in mainstream OO languages. This is another kind of expressiveness aspect of the X/O impedance mismatch. An XML programmer (used to say XQuery) would be ‘disappointed’ when moving to Java or C# (assuming a classic X-to-O mapping). Here is a comprehensive list of XML axes that are missing in OO:

- Attribute axis
- Parent axis
- Descendant/Descendant-or-self axes
- Ancestor/Ancestor-or-self axes
- Following-/Preceding-sibling axes
- Following/Preceding axes (i.e., nodes following/preceding in document order)

Also, XPath provides a general idiom for expressing *filters* on axes.

Fig. 15 shows another (somewhat contrived) encoding for totaling orders; it leverages both the descendant and parent axis. One could argue that this example is more ‘adaptive’ in that the use of the descendant axis helps us to detach ourselves from the precise shape of the XML tree; we only commit to the fact that there are items and that zip codes are found at the parents of items — no matter how many levels we need to descend to find items; no matter what the element name may be for the parent of items.

There are several OO programming techniques and language extensions that make a related contribution. Adaptive programming [35], which has been embedded into various OO languages, provides an efficiently executable selector language with coverage of an idiom that is reminiscent of the descendant axis. Similarly, there are advanced visitor techniques and OO embeddings of term-traversal strategies [63,34]. The C#-like research language $C\omega$ [42,41,6] provides a primitive (“...”) that mimics the descendant axis. When faced with general (potentially cyclic) object graphs, it is actually not obvious what the behavior of such an axis would be.

3.5 Node labels vs. edge labels

Element names are essentially node labels whereas property names are essentially edge labels. An element name is semantically part of the element itself, whereas an OO member name (of a field or a property) is a selector for a subcomponent. In fact, objects are also labeled — by means of a type tag. While these type tags seem to be similar to element names, they do not serve any established purpose for member selection. Mainstream technologies for X-to-O mappings tend to ‘neglect’ this difference. A common XSD mapping rule (used by various X-to-O mapping tools) reads as follows.

When mapping a content model, given a local element declaration with element name n and element type t , the corresponding field declaration leverages n as name and t as type.

In Fig. 16, we illustrate the intuition that goes with the above mapping rule. The first schema style really ‘makes one think’ that element particles are like field declarations. However, semantically, the local element declarations define local types of elements with the local element names as intrinsic parts. The second schema style is perhaps less misleading: the types of elements are ‘prefabricated’ in global declarations and then merely referenced in content models. The two formulations define the same XML language (modulo some fine details).

It should be clear by now that the choice of the (mapped) element type (cf. “type =...” in an element declaration) as the type of the associated field or property implies imprecision. Consider the XPath expression `item/price` which selects all `price trees` as opposed to `price values`. Hence, mapping the element particle `price` to a field of type `double` is sloppy. One could say that this sort of mapping rule *composes node selection and value access*. In any case, there are some potentially harmful implications:

- We effectively lose information: the element name. If we later inspect a `price` in the form of a value of type `double`, it is impossible to observe that it is actually a price.
- It follows that type equivalence for elements (such as prices) is relaxed to type equivalence for element types (such as doubles).
- We also lose the parent axis — at least for simple element types because we cannot possibly attach any parent pointer to values of primitive types.

With some effort, we can preserve XML’s node labels in the object world. To this end, we need to designate distinct object types to all global and local element declarations.

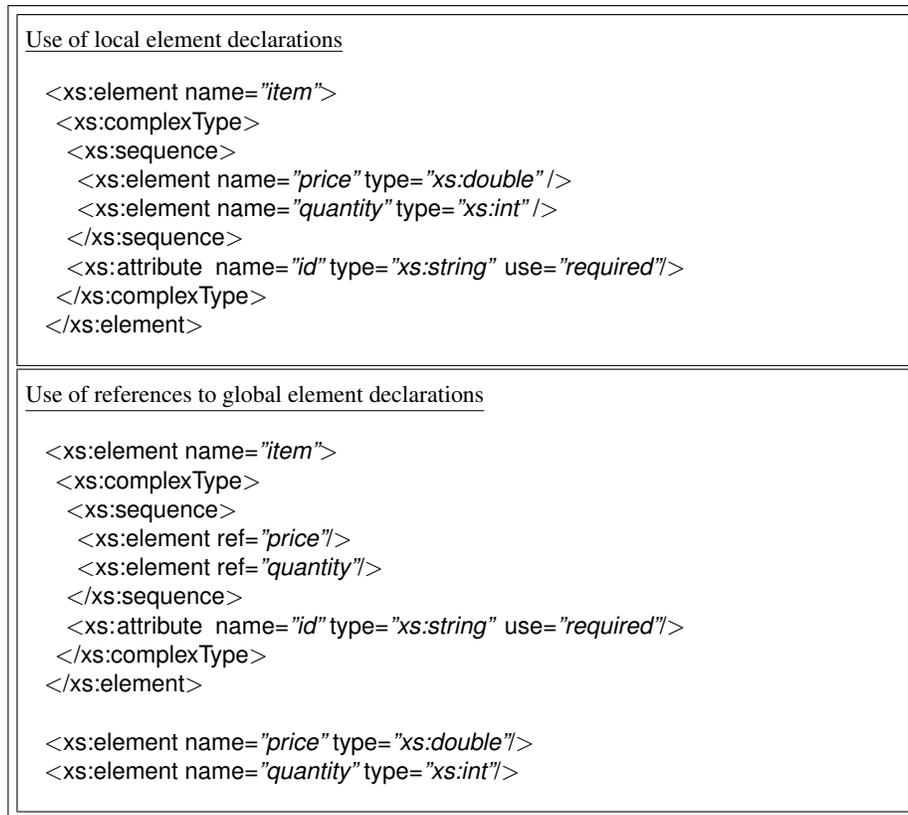


Fig. 16. Different styles of schema organization

(For the sake of a homogeneous situation, we may also designate object types to attribute declarations.) Despite the use of element names as type names, the element names may continue to serve as property names (modulo qualification problems for element names, as discussed later).

When we apply the new rule to the schema for purchase orders, we end up with the following additional object types: `id`, `zip`, `price`, `quantity`; cf. Fig 17. The new types are defined as ‘wrapper types’; they wrap content of the appropriate types, i.e., `double`, `int` or `string`. The wrapping idiom is factored out to a generic class `Wrapper`. This class also provides an implicit cast operation for wrapper access.¹⁴ The implicitness of the cast implies that the original query code for computing the total does not need to be

¹⁴ A note on C#: User-defined cast operators are very much like static methods except that they mediate between two types. There is the **explicit operator** form which can be seen as a user-defined down-cast as in `(double)aPriceObject`. There is also the **implicit operator** form; implicit operators are automatically applied, just as up-casts, whenever the context of an expression requires the target type. In both cases, the two involved types must not engage in subtyping relationships so that the extra casts do not introduce any ambiguities.

<p><u>Node-labeled object types of a complex types</u></p> <pre> public class orders { public order[] order; } public class order { public item[] item; public id id; public zip zip; } public class item { public price price; public quantity quantity; public id id; } </pre>
<p><u>Node-labeled object types of a simple types</u></p> <pre> public class id : Wrapper<string> { } public class zip : Wrapper<int> { } public class price : Wrapper<double> { } public class quantity : Wrapper<int> { } </pre>
<p><u>Generic helper class</u></p> <pre> public class Wrapper<T> { public T Value; public static implicit operator T(Wrapper<T> it) { return it.Value; } } </pre>

Fig. 17. Illustration of a node-label-preserving mapping

changed in a single detail. Hence, the increased precision of the new object model does not negatively affect query convenience.

This development triggers a question:

*Why is such a node-label-preserving mapping not commonly used?
It appears to be the only sensible option from a conceptual point of view!*

Here is an attempt of an explanation:

- The mapping is not too obvious in the first place.
- Field declarations of the form “**public** price price;” are arguably unpalatable.

- Programmers expect the systematic use of familiar value types (such as `double`).
- The weaknesses of a more sloppy mapping have not been discussed thoroughly.
- The mapping of local element declarations may lead to heavily nested object types.
- The additional ‘object types’ may be overwhelming and imply overhead.
- Object construction may require construction of many ‘mini objects’.

We will regularly return to the tension between node-label omission and preservation. We will further substantiate that the (common) node-label-omitting mapping is indeed inferior in a conceptual sense, but the (uncommon) node-label-preserving mapping is challenged by palatability considerations and technical problems.

3.6 Ordered vs. unordered edges

The edges of an XML tree are *positionally* labeled, thereby expressing significance of order among the immediate subtrees of an XML tree. This order is programmatically relevant in various ways (in the context of XPath-like XML processing). For instance, there are sibling axes. Most importantly, there is the notion of *document order* that defines the order of nodes in query results.

In contrast, the edges in an object graph are *nominally* labeled while the order of these labels (such as the textual order in the program text) does not matter, as far as the idealized, mathematical semantics is concerned. (It may matter with regard to performance due to object layout. It may also be observable by reflection.) In fact, XML *attributes* are nominally labeled, too.

When performing canonical X-to-O (and O-to-X) mappings, it is common to identify textual order of OO member declarations with XSD’s `<sequence>` composites. When XML data is de-serialized into objects, then it is common to be flexible with regard to order. That is, an object is populated with subtrees even if they appear in an order that is different from the schema-prescribed `<sequence>` composite. One argument in favor of such behavior is that the added flexibility increases interoperability.

`<all>` composites¹⁵ are typically mapped just as if they were `<sequence>` composites (while de-serialization would be more relaxed, if it was order-aware for `<sequence>` composites). However, one could argue that the order of children in the input should be maintained by populated objects (for the purpose of round-tripping; also order may matter). Likewise, an object type for an `<all>` composite should admit different orders of populating the composite. Such a behavior cannot be expected from plain objects.

Fig 18 demonstrates a typical form of treating *order along de-serialization and serialization*. The class at the top is the ‘native’ object type from which we start. The schema in the middle of the figure has been derived by `xsd.exe` (used in the O-to-X direction). We see that a `<sequence>` composite is set up. Here it is assumed that the textual order of member declarations in a serializable class may indeed be a hint at the preferred serialization order. Also, XSD’s `<all>` cannot be generally used anyhow

¹⁵ A note on XSD: there is the `<all>` compositor used for the construction of content models. The compositor expresses that the components may occur in any order as opposed to the sequential order of the `<sequence>` compositor. (`<all>` groups are reminiscent of ‘permutation phrases’ for string grammars [13].) One may think that such grouping expresses deviation from ordered edges, but it rather expresses *insignificance of order during validation*.

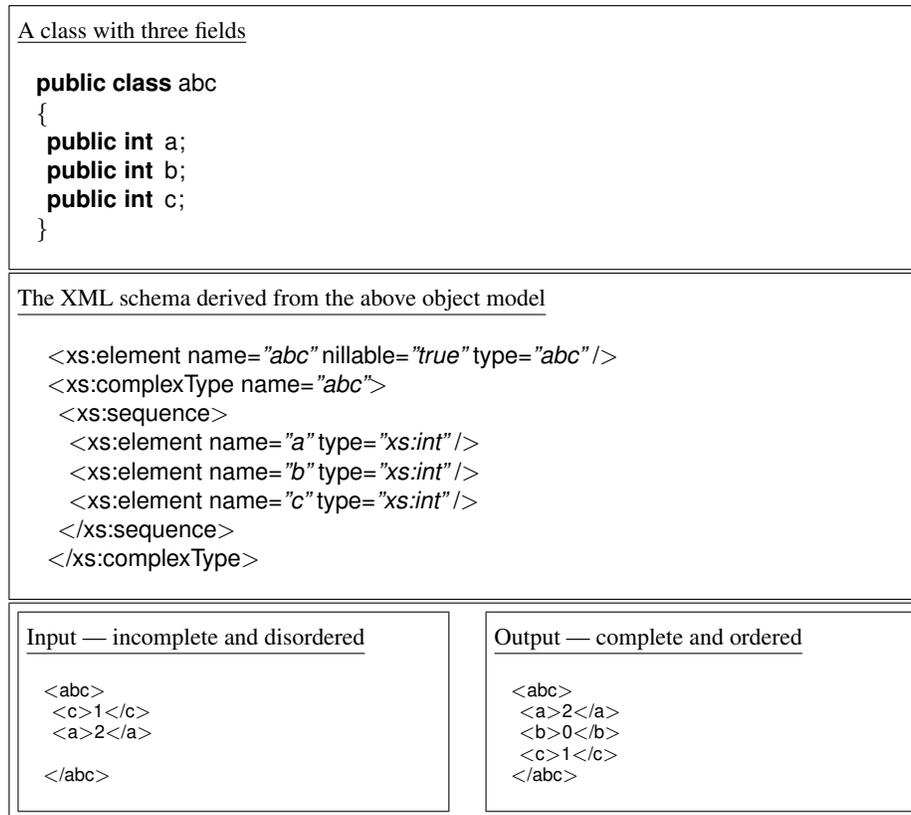


Fig. 18. A test case for ordering and presence

because of XSD 1.0 expressiveness limitations. Despite the commitment to `<sequence>`, de-serialization may still be more liberal with regard to order (and therefore effectively handle the `<sequence>` like an `<all>`).

The input/output pair in the figure shows the behavior for de-serialization followed by immediate serialization. Hence, disordered content is accepted, and defaults are inferred for absent subtrees. One may argue that these are actually two independent issues: *flexible handling of order vs. default values for missing subtrees or attributes*. One may also argue that the violations should be subjected to a relaxed validation scheme to be demanded explicitly by user code.

3.7 Qualified vs. local selectors

At the value level, the element names (labels) in XML may be unqualified or qualified, but there are also rules for a sort of implicit qualification. At the type level, element names may be unqualified or qualified. The potential of qualification stresses the mapping of selectors in the XML sense (i.e., element names) to selectors in the OO sense (i.e., OO member names) because OO member names are always local — relative to

an object type. Here are typical options for dealing with potentially qualified element names; all of them leave a bad aftertaste:

1. Ignore qualifiers; adopt name mangling for disambiguation, if necessary.
2. Append namespace prefixes to member names for disambiguation, if necessary.
3. Drop into an untyped, DOM-like representation.

Fig. 19 shows a ‘test case’ for qualified element names. We define a content model with three different *bar*’s. The first *bar* comes from the target namespace of the schema at hand; cf. prefix *ins*. The second *bar* is contributed by a local element declaration. The third *bar* is imported from the namespace with prefix *ins*. The options ‘ignore qualifiers’ and ‘append namespace prefixes to member names’ are illustrated in Fig. 19. (The option ‘ignore qualifiers’ is adopted by the `xsd.exe` tool where name mangling appends an index “1”, “2”, etc. to a field name, if disambiguation is required. JAXB [54] adopts the option to ‘drop into the DOM’ for content models of certain shapes, including the one used in the test case.)

So we should reconsider the idea of leveraging element names as OO member names. We may use a different protocol for member access: *type-driven OO member access*. Conceptually, element references are indeed similar to type references as one is used to in OO programming languages — including the possibility of qualification for such type references. However, a type-driven access protocol results in an ‘unusual’ programming model. The type-driven protocol is sketched in App. A.1; it turns out to require cumbersome encoding efforts.

3.8 Semi-structured vs. structured content

A ‘plain object’ for data representation can be viewed as a dictionary that maps field names to values. As we have discussed, node-labeled XML trees do not fully align with this view (due to the lack of unambiguous, nominal selectors). Even more seriously, there is no straightforward OO counterpart for semi-structured content, i.e., content that intersperses elements and text. XML also admits additional components such as *XML comments and PIs*. Hence, we face a representation challenge: what sort of X-to-O mapping do we devise so that the additional XML-isms do not get lost. In addition, we would also want a reasonable programming model such that the XML-isms can be accessed in queries and updates.

Let us focus on interim text in the sense of mixed content. (XML comments and PIs require similar efforts.) Fig. 20 shows an XML document with a letter that involves mixed content; the figure also shows a corresponding schema fragment; cf. `mixed="true"`, as well as the mapping result obtained with the `xsd.exe` tool. The schema-derived class devises a field, `Text`, that stores interim text in a string array. At the bottom of the figure, we also show the letter as it looks like after a de-serialization + serialization sequence. As we can see, the XML data does not round-trip; all interim text is appended to the child elements.

In general, there are the following options for handling mixed content:

1. Drop into a DOM-like representation.

An XML schema with an import

```

<xs:schema targetNamespace="http://tempuri.org/foo"
  xmlns:tns="http://tempuri.org/foo"
  xmlns:ins="http://tempuri.org/bar"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:import namespace="http://tempuri.org/bar"/>

  <xs:element name="foo">
    <xs:complexType>
      <xs:sequence>
        <!-- Three different bar's -->
        <xs:element ref="tns:bar" />
        <xs:element name="bar" type="xs:int"/>
        <xs:element ref="ins:bar"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="bar" type="xs:string"/>

</xs:schema>

```

An imported XML schema

```

<xs:schema targetNamespace="http://tempuri.org/bar"
  xmlns:tns="http://tempuri.org/bar"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="bar" type="xs:double"/>

</xs:schema>

```

Mapping option: names without hints at qualifiers

```

public class foo
{
  public string bar;
  public int bar1;
  public double bar2;
}

```

Mapping option: names with namespace prefixes appended to them

```

public class foo
{
  public string barTns;
  public int bar;
  public double barIns;
}

```

Fig. 19. Mapping qualified element names

<p><u>A semi-structured letter</u></p> <pre><letter> Dear Mr.<name>Foo Smith</name>. Your order <id>8837</id> will be shipped on <shipdate>2008-04-01</shipdate>. </letter></pre>
<p><u>A schema for semi-structured letters</u></p> <pre><xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"> <xs:element name="letter"> <xs:complexType mixed="true"> <xs:sequence> <xs:element name="name" type="xs:string"/> <xs:element name="id" type="xs:unsignedInt"/> <xs:element name="shipdate" type="xs:date"/> </xs:sequence> </xs:complexType> </xs:element> </xs:schema></pre>
<p><u>A possible mapping result</u></p> <pre>public class letter { // Fields for element nodes public string name; public uint id; public System.DateTime shipdate; // Interim text public string[] Text; }</pre>
<p><u>Serialization result</u></p> <pre><letter> <name>Foo Smith</name> <id>8837</id> <shipdate>2008-04-01</shipdate> Dear Mr.. Your order will be shipped on . </letter></pre>

Fig. 20. Illustration of mixed content models

```

public class letter
{
    // State as a plain list of nodes
    public List<object> nodes = new List<object>();
    // Getters as type-driven filters
    public name name      { get { return nodes.OfType<name>().FirstOrDefault(); } }
    public id id         { get { return nodes.OfType<id>().FirstOrDefault(); } }
    public shipdate shipdate { get { return nodes.OfType<shipdate>().FirstOrDefault(); } }
}

// Node-label-preserving object types for simple-typed elements
public class name      : Wrapper<string> { }
public class id       : Wrapper<uint>  { }
public class shipdate : Wrapper<DateTime> { }

// Generic helper class
public class Wrapper<T>
{
    public T Value;
    public static implicit operator T(Wrapper<T> it) { return it.Value; }
}

```

Fig. 21. Mixed content preservation along round-tripping

2. Store text separately from the fields for child elements.
3. Provide a collection of interim text and typed objects.

X/O mapping technologies exercise all these options. As we have illustrated, the second option is chosen by `xsd.exe`; cf. Fig. 20. JAXB (ever since version 1) favors the third option, where the access to the heterogeneous collection (say, list) is provided by a so-called ‘general content property’. This approach, when compared to ‘dropping into the DOM’, is more typed because the items in the list (besides text nodes) are still of schema-derived object types as opposed to a generic element type. A general content property provides less static typing than regular OO members for child elements. In particular, a general content property makes it easy to construct a node list that violates the relative order constraints defined by a `<sequence>` composite.

Ultimately, one may want to combine the strengths of a ‘general content property’ (which is convenient for observing and constructing mixed content) and OO members for the child elements. We attempt such a combination for the letter example in Fig. 21. The design can be summarized as follows. We store all content in a plain list, `nodes`, thereby maintaining the order of element and text nodes. The properties for the child elements operate on the `nodes` list in a type-driven manner. (We only show getters in the figure; setters are more complicated because of order constraints.) Without loss of generality, we assume a node-label-preserving mapping. (A node-label-omitting mapping would require extra ‘housekeeping state’ to maintain labels aside.) ‘XML objects’, as discussed in Sec. 2.4, would naturally provide both, a general content property and OO members for the child element. However, the present discussion alludes to the relative complexity of such an implementation — also taking into account arbitrary XSD patterns and assuming full-fledged getter/setter functionality.

3.9 Tree literals vs. object initialization

XML trees (say, XML literals) are constructed by essentially listing subtrees (and other nodes) at each level. An XML literal is the complete and direct representation of data as opposed to any form of private state in OO programming. In contrast, objects are constructed by designated (implicitly or explicitly defined) constructor members that initialize the state of a new object in a programmer-defined manner, potentially taking into account constructor arguments. Also the state space of an object type is typically exercised by calling methods. Despite this fundamental difference, one would hope that ‘nested object construction’ is capable of simulating the construction of XML literals. Unfortunately, the means for static type checking of object construction are too weak to rule out the construction of invalid XML content. (Arguably, our discussion starts to shift from ‘data models’ to ‘type systems’.)

Fig. 22 illustrates programmatic XML-tree construction in different styles. As a baseline, at the top, we construct an untyped XML tree using the functional element constructor of the LINQ to XML API; cf. `new XElement(...)`. This form follows closely the shape of the original XML data. That is, the various attributes and child elements of orders or items are passed as arguments to the constructors.

The middle and the bottom parts in the figure illustrate typed construction assuming two different mappings and different idioms for construction. The code in the middle relies on a node-label-omitting mapping. Also, we employ the expression-oriented syntax for object initialization of C# 3.0. To clarify this construct, let us expand the expression-oriented syntax for one item of a purchase order:

```
// Object– initialization syntax
var i = new item { id = "23", price = 42, quantity = 2 };

// Expanded form
var i      = new item();
i.id      = "23";
i.price   = 42;
i.quantity = 2;
```

The code at the bottom of the figure relies on a node-label-preserving mapping. Also, we use ‘functional constructors’ to obtain an expression-oriented style of nested tree construction. We did not show these constructors before, but they are trivially defined, given the simplicity of the schema for purchase orders, as shown in the figure.

At first sight, both of the methods for typed construction seem to be quite reasonable. However, there are several issues that should be called out:

- OO constructors are more ‘type-oriented’ than ‘instance-oriented’. That is, the objects that correspond to the repeating item element must be grouped as a collection. (In contrast, the item elements would be part of a flat list of children of the order element — when forming an XML literal.) We want to argue that an OO programmer may actually appreciate this deviation from XML style.
- Type checking for the object-initialization syntax does not account for occurrence constraints and the constraints implied by the different compositors. For instance, type checking does not establish that all ‘required’ sub-objects are determined.

Construct an untyped, nested XML tree

```

new XElement("order",
  new XAttribute("id", "47"),
  new XAttribute("zip", "98052"),
  new XElement("item",
    new XAttribute("id", "23"),
    new XElement("price", "42"),
    new XElement("quantity", "2")),
  new XElement("item",
    new XAttribute("id", "32"),
    new XElement("price", "33"),
    new XElement("quantity", "3")));

```

Object initialization for a node-label-omitting mapping

```

new order {
  id = "47",
  zip = 98052,
  item = new item[] {
    new item { id = "23", price = 42, quantity = 2 },
    new item { id = "32", price = 33, quantity = 3 } };

```

Functional construction for a node-label-preserving mapping

```

new order(
  new id("47"),
  new zip(98052),
  new item[] {
    new item(new id("23"), new price(42), new quantity(2)),
    new item(new id("32"), new price(33), new quantity(3)) } );

```

Relevant functional constructors

```

public order(id id, zip zip, item[] item)
{
  this.id = id;
  this.zip = zip;
  this.item = item;
}

public item(id id, price price, quantity quantity)
{
  this.id = id;
  this.price = price;
  this.quantity = quantity;
}

```

Fig. 22. Different construction styles

- Functional constructors are convenient for sequences of mandatory sub-objects. As soon as we have to deal with choices or even nested composites, we would need multiple constructors. The approach does not scale for optional particles, <all> composites, and attributes.
- The use of functional constructors requires the (uncommon) node-label-preserving mapping for the benefit of program comprehension. That is, XML literals systematically identify element labels for all subtrees whereas the node-label-omitting mapping would not provide any counterparts for the element labels.
- Object initializers cannot be used for content models with recurrent element names (unless we assume an object initializer to make multiple contributions to a given sub-object of a collection type). Functional constructors naturally deal with this problem since multiple argument positions can be of the same type.
- Both techniques equally fail on mixed content and XML comments.

App. A.2 engages in a heavy encoding showing that, in principle, one can provide a construction protocol solving most of the above problems. We use a technique that encodes content models as finite-state machines *at the type level*. Arguably, the encoding is too convoluted for practical X-to-O mappings.

One may argue that language support for regular expression types [27,22] would allow us to better type-check object construction. Such type-system extensions are not available in mainstream languages such as C# and Java. It is important to note that plain, regular expression types are insufficient to cover a comprehensive XML-type system like XSD. Think of subtyping; think of the dichotomy elements vs. attributes. Also, <all> composites and attribute collections are not compatible with basic regular expression types (because these constructs abstract from order). Furthermore, mixed content would require extra expressiveness. Finally, the effective use of regular expression types would generally require the (uncommon) node-label-preserving mapping because all type distinctions from the content models would need to be observed by the regular expression types.

4 The X/O type systems

We will now look into aspects of the X/O impedance mismatch that involve XML types and object types in a more intimate manner, even though many of the aspects are grounded in the differences between the data models of XML and objects. The following subsections discuss a number of challenges offered by XML schemas:

1. Occurrence constraints
2. Choice types
3. Nested composites
4. Local elements
5. Element templates
6. Type extension
7. Element substitution
8. Type restriction
9. Simple types

A small part of the discussion is specific to (idiosyncrasies of) XSD.

Optional elements both of simple and complex types

```

<xs:element name="opts">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="a" type="xs:int"/>
      <xs:element name="b" type="xs:int" minOccurs="0"/>
      <xs:element name="c" type="foo"/>
      <xs:element name="d" type="foo" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="foo"> ... details elided ... </xs:complexType>

```

A node-label-omitting mapping

```

public class opts
{
  public int a; // Required field
  public int? b; // Uses a .NET nullable type
  public foo c; // Required field
  public foo d; // Rely on null reference
}
public class foo { ... }

```

A node-label-preserving mapping

```

public class opts
{
  public a a;
  public b b;
  public c c;
  public d d;
}

public class a : Wrapper<int> { }
public class b : Wrapper<int> { }
public class c : Wrapper<foo> { }
public class d : Wrapper<foo> { }

public class foo { ... }

```

Fig. 23. Mapping of optionality

4.1 Occurrence constraints

We will focus here on optionality and mandatoriness. The discussion generalizes in a unsurprising manner for arbitrary `minOccurs/maxOccurs` constraints — in particular for possibly empty, non-empty and bounded repetitions of elements.

XSD offers regular expression-like optionality; cf. `minOccurs="0"` (and `maxOccurs="1"` per default). Mainstream programming languages like C# and Java provide optionality for reference types *by default* since these types are nullable, i.e., their value domains comprise the null reference. This implies that optionality is covered ‘automatically’ by an X-to-O mapping, except for fields or properties of value types. We can explicitly express optionality for value types by using an appropriate type constructor; in fact, .NET readily provides nullable (value) types.¹⁶

Fig. 23 shows an XML schema that exercises optionality for simple and complex element types, while required elements are also declared for comparison. Both a node-label-omitting and a node-label-preserving mapping are exercised. (Again, we use the `Wrapper` class from Sec. 3.5.) The first option illustrates the asymmetry of implicit nullability for reference types and explicit nullability for value types. The second option only uses reference types, and hence the asymmetry vanishes — at the cost of losing all discoverability of optionality. That is, optional and mandatory particles both end up as implicitly nullable members. (The schema constrains may still be enforced through run-time checks of properties.) The lack of discoverability is worrying. For instance, during object access, members of implicitly nullable types do not tell the programmer which parts of an object may be perfectly missing and hence require a programmatic case discrimination for optionality. Tool tips and other IDE techniques may help to mitigate this problem.

In principle, one could adopt a discipline such that optionality is generally made explicit (not just for value types). We defer such an experimental treatment of nullability to App. A.3. Such a degree of explicitness may be potentially considered as unpalatable.

An idiosyncrasy: nillability

XSD complements optionality with nillability — a way of saying that ‘an element is there and not there’. That is, by setting `xsi:nil` to `true` on an instance element, the content of the element can (in fact, must) be omitted. However, the empty element can carry attributes as prescribed by its schema type. An X-to-O mapping can deal with nillability in these ways:

- Infer `xsi:nil` from absence of content.
- Provide a property to set `xsi:nil`.

Unfortunately, feature interaction complicates the matter. That is, there are potentially hybrids: ‘niloptables’, i.e., nillable, optional particles:

¹⁶ A note on .NET nullable types: The type constructor `Nullable` (also denotable as “?” in C#) can only be applied to a value type, as opposed to a reference type. It provides observations `HasValue` to check for a non-null value and `Value` to extract a non-null value.

```

<xs:element name="niloptable">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="foo" type="xs:int" minOccurs="0" nillable="true"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

So we may need to distinguish ‘omitted in the sense of optional’ vs. ‘present but nil’. Fortunately, nillability, even without optionality, is used relatively seldom according to a study [32] — at least in the case of hand-crafted schemas. However, we have seen components for schema generation (or export) that systematically use nillability.

Enforced mandatoriness

Since optionality is the default for reference types, we may want to enforce mandatoriness; cf. `minOccurs="1"`. In particular, we may check at run-time that no nulls are passed to setters for mandatory particles, and no nulls are returned by the getters. For instance, the mandatory element particle `c` would be implemented as a property as follows:

```

public foo c {
  get { if (.c==null) throw ...; return .c; }
  set { if (value==null) throw ...; .c = value; }
}
private foo _c;

```

As long as we admit default constructors, there is no guarantee that objects are fully defined with regard to mandatory particles; hence the getters should indeed perform checks. The elimination of default constructors would be a radical step; OO programmers are used to the idea of initializing instances and defining sub-objects incrementally. More importantly, a general scheme of non-default constructors is not in sight, as we discussed in Sec. 3.9.

This also leads us to review non-nullable types [18], as they are becoming available through extensions of C# and Java. As demonstrated by $C\omega$ [42,41,6], non-nullable types can also be seamlessly integrated with regular expression types. In the following experiment, we make mandatoriness for reference types explicit:

```

public class opts
{
  public int a; // required element of simple type
  public int? b; // optional element of simple type
  public foo! c; // required element of complex type
  public foo d; // optional element of complex type
}

```

We assume the type constructor “!” for non-nullable types. In the context of X-to-O mapping, we need a form of non-nullable types that admits ‘transient nulls’ (as part of a

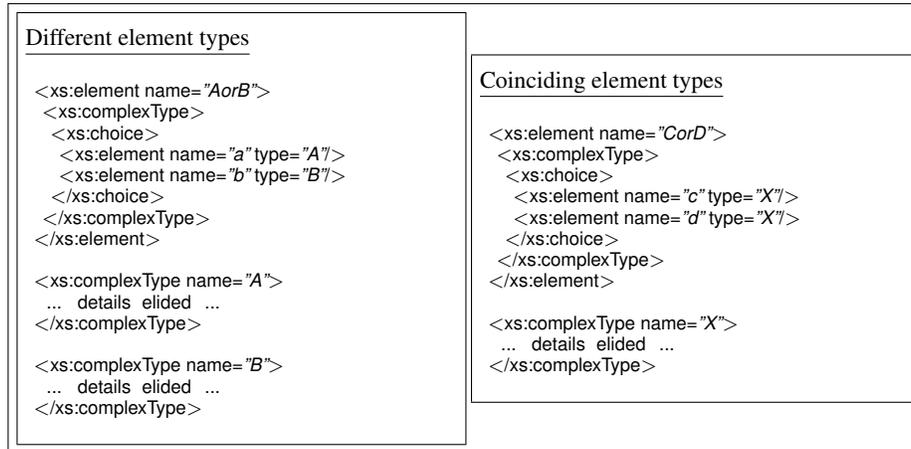


Fig. 24. Different kinds of choices

stepwise object initialization protocol) or nulls due to ‘slightly invalid’ content. Active assignment of nulls should be prohibited by the type system, but fields of non-nullable types should be allowed to hold null, as long as they are not accessed (say, read). Hence, such non-nullable types would still involve some degree of dynamic checking.

4.2 Choice types

Choice types (such as XSD’s `<choice>` composites or DTD’s form “*a|b*”) provide a special kind of discriminated union, in fact, *type-discriminated unions*, also known as *type-indexed co-products* (TICs; [53]).¹⁷ Choice types or TICs are not natively available in OO mainstream languages. Type-system extensions for regular-expression types cover some form of choices. The research language *C ω* [42,41,6] integrates choice types into an otherwise C#-like type system (without updates though). For OO languages like Java and C#, the goal must be to devise a mapping for choice types by encoding them in some way.

For simplicity, we focus here on choices over plain element particles, deferring the discussion of nested composites until Sec. 4.3. Fig. 24 exercises two principled patterns for choices. On the left-hand side, we see a choice where the element types (A vs. B) are different. On the right-hand side, the element types coincide; only the element names are different. The latter pattern is more problematic in the context of the (common) node-label-omitting mapping, as we will demonstrate shortly.

The simplest kind of mapping for choices may look as follows:

```

public class AorB
{
  public object Item;
}

```

¹⁷ For the record, TICs [53] are more general than XSD choices in so far that they require pairwise distinct branch types but not the stricter UPA condition of XSD [67].

That is, a single `Item` field stores the value of the choice. The type of `Item` is the least upper bound of the participants in the choice; this bound tends to be `object`. Case discrimination must be based on instance-of tests. Neither the type system nor ‘IntelliSense’ can be expected to be helpful in picking reasonable types for these tests.

The mapping for the second sample must devise an extra tag field since the mapped element types by themselves do not admit discrimination. (The shown mapping results for `AorB` and `CorD` were obtained with the `xsd.exe` tool.)

```
public class CorD
{
    public X Item;
    public ItemChoiceType ItemElementName;
}
public enum ItemChoiceType { c,d }
```

There are obvious problems with the options discussed so far:

- We can store objects of inappropriate types in the slot for the choice.
- The tagged union is unsafe because the tag and the item are set independently.
- Object construction does not enforce population of either branch of the choice.
- Code for case discriminations may (accidentally) omit cases.

We can definitely improve on all these concerns, but the resulting mapping will become increasingly more complex. Also, several improved mapping options for choice types (that one can think of) rely on extra preconditions on the XSD patterns that involve `<choice>`. Ultimately, we seek a simple and general treatment of choices.

As an experiment, Fig. 25 shows a conceptually appealing mapping:

- The actual representation of the choice’s value is opaque.
- The branch can be queried based on a read-only tag.
- Injection is facilitated by implicit cast and construction.
- Projection is facilitated by explicit cast.

This scheme suffers from several limitations: (i) while the first sample can be mapped in this manner, the second sample would require the (uncommon) node-label-preserving mapping so that sufficient type distinctions can be used by the casts; (ii) nested composites cannot be mapped in this manner, unless we somehow introduce (named) classes for the inner composites; (iii) the resulting programming model does not align with XML programming practice because there is no XPath-like child axis.

We could fix the last problem by adding query members as follows:¹⁸

```
public A a { get { return any as A; } } // set omitted
public B b { get { return any as B; } } // set omitted
```

¹⁸ A note on C#: the “`e as T`” construct essentially behaves as follows: given the value of `e`, its type is checked to be a subtype of `T`, and if so, the value is casted to `T`; otherwise the expression form evaluates to `null`.

```

public class AorB
{
    // Injections (by construction)
    public AorB(A v) { any = v; _typeCode = TypeCode.a; }
    public AorB(B v) { any = v; _typeCode = TypeCode.b; }

    // Injections ("up-cast")
    public static implicit operator AorB(A v) { return new AorB(v); }
    public static implicit operator AorB(B v) { return new AorB(v); }

    // Projections ("down-cast")
    public static explicit operator A(AorB anOr) { return (A)anOr.any; }
    public static explicit operator B(AorB anOr) { return (B)anOr.any; }

    // Tag + inspection
    public enum TypeCode { a,b };
    public TypeCode typeCode { get { return _typeCode; } }

    // Private state
    private object any;
    private TypeCode _typeCode;
}

```

Fig. 25. Choices based on casts

```

public class CorD
{
    // Getters and setters
    public X c {
        get { return (_typeCode==1)?any:null; }
        set { any=value; _typeCode=1; }
    }
    public X d {
        get { return (_typeCode==2)?any:null; }
        set { any=value; _typeCode=2; }
    }

    // Private state
    private X any;
    private int _typeCode;
}

```

Fig. 26. Choices as sequences

We may also add setters that ‘alter’ the choice. We may further add a default constructor that sets up an uncommitted choice. Hence, the getters may return null. Incidentally, the getters and setters are expressive enough to operate on the choice, and hence we

can eliminate the original injection/projection protocol; cf. Fig. 26 for the end result. In fact, we have obtained a general and relatively convenient mapping rule for choices. Admittedly, choices and sequences cannot be distinguished anymore in terms of the interface of a schema-derived class. (Only the behavior of the properties is different; the setters for a choice implement a mutual exclusion protocol.) Tool tips and other IDE techniques may restore some degree of discoverability for choices.

4.3 Nested composites

‘Flat’ composites can be modeled by object types that plainly aggregate members. (Recurrent element names require more effort; cf. Sec. 3.3.) Nested composites cannot be modeled by nested object types (classes), unless we invent names for the inner composites. Here is a representative list of nesting patterns as they can be used in XSD (and other XML-schema languages); we use regular expression notation for conciseness:

element	choiceInSequence	= a (b c)
element	sequenceInChoice	= a (b c)
element	plusOnSequence	= (a b) ⁺
element	plusOnNestedSequence	= a (b c) ⁺
element	plusInAndOut	= a (b ⁺ c) ⁺

In the terminology of grammars, the above patterns exercise expressiveness of EBNF. Mainstream X-to-O mapping technologies tend to be challenged by these patterns. In some cases, the schema-derived object models drop into untyped representations (DOM and friends); in other cases, the round-tripping requirement is violated. We can identify the following overall options for mapping nested composites:

1. Reject nested content models.
2. Model compositors as OO generics.
3. Introduce auxiliary classes for anonymous composites.
4. Relax nested content models so that they become flat.
5. View nested content models in a flat manner.

We discuss the details of these options in the sequel.

Reject nested content models

This option is too limiting. Nested content models are frequently used in real-world schemas [32] according to a study. One may think that the XML programmer should be responsible to refactor the schema until all content models are flat. However, the benefits of *typed* XML processing and X-to-O mapping should be attainable without putting such a burden on the XML programmer.

Model compositors as OO generics

Based on suitable generics, we can enable the structural composition of OO types in a way that parallels the composition of XSD composites. That is, we need generics

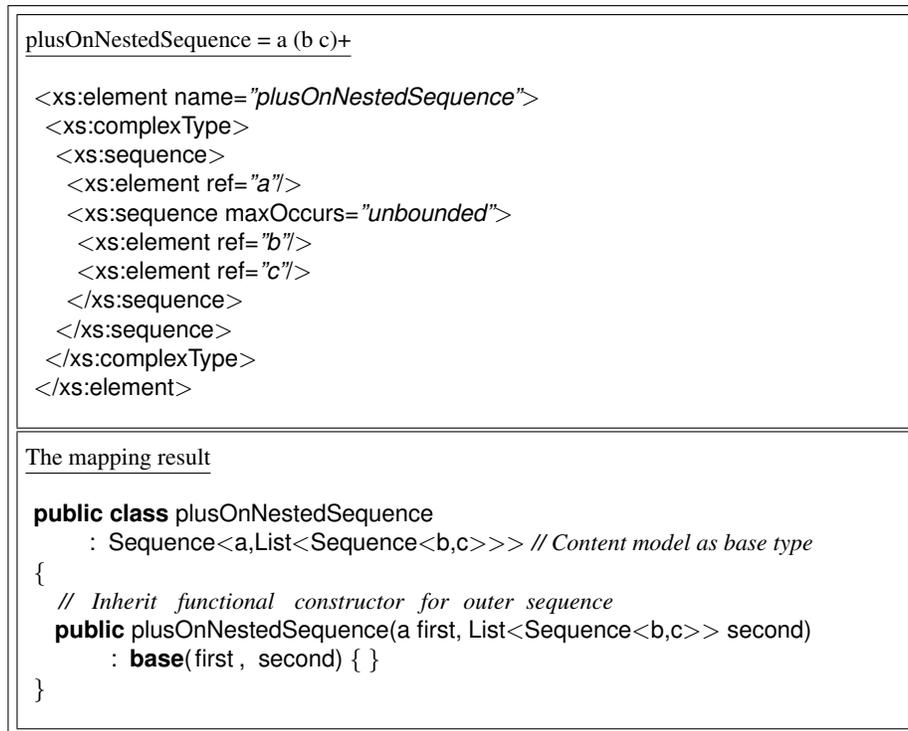


Fig. 27. Map a nested content model using generics for composites

for `<sequence>` and `<choice>` (perhaps also for `<all>`, which we ignore here because it is hardly used [32], according to a study, and hard to support faithfully). The discussion of choice types has alluded to a suitable model for the compositor `<choice>`. The compositor `<sequence>` is conceptually trivial; it corresponds to a product-type constructor in the sense of universal algebra. For completeness' sake, we include suitable generics in App. A.4. Fig. 27 illustrates the use of generics (including the `List<...>` class) for a non-trivial nesting pattern. The schema-derived object type simply inherits from the composed structural type. This approach suffers from a number of problems:

- The ‘algebra of regular expressions’ is not observed by compound generics expressions. For instance, in the algebra of regular expressions, we have $(x^*)^* = x^*$ while in the typical object-type system, we have `List<List<x>> ≠ List<x>`.
- If choices with type-driven injections and projections are favored, then the (uncommon) node-label-preserving mapping is needed. We may consider position indexes instead, at the cost of making the programming model less convenient.
- We need generic classes `Sequence<...>` and `Choice<...>` for all arities 2, 3, 4, ... The available polymorphism in mainstream OO languages does not allow us to parameterize over arity. As a remedy, we may switch to the nested use of the binary generic classes, again, at the cost of making the programming model less convenient.

- Using the structural type as the superclass of a schema-derived class rules out the use of class inheritance for schema-defined substitutability, if we assume single class inheritance. As a remedy, we may restrict the use of generic classes to inner (i.e., truly anonymous) composites, which however implies a naming challenge for top-level members that grant access to inner composites.
- Manipulation of nested generics is known to be inconvenient in mainstream OO languages [23] — due to limitations of type inference and other issues. Also, as a matter of palatability, OO developers are used to object navigation based on named members as opposed to positions or types.

Introduce auxiliary classes for anonymous composites

We may attempt to systematically name all anonymous composites. As a result, we could use (nested) object types to model the nested composites. As a result, familiar member-based access can be used (as opposed to the positions or types in the case of the previous option). As an experiment, we suggest some nomination rules:

- Use the following local type names for nested composites:
 - **Sequence**, **Sequence2**, ... for `<sequence>`.
 - **Choice**, **Choice2**, ... for `<choice>`.
- Use the following member names for accessing composites:
 - **AsSequence** for non-repeating `<sequence>`.
 - **AsChoice** for non-repeating `<choice>`.
 - **AsList** for a repeating `<sequence>` or `<choice>`.
 - * Return type:
 - **List<Sequence>**, ... for repeating `<sequence>`.
 - **List<Choice>**, ... for repeating `<choice>`.

We illustrate these rules in Fig. 28. Additional refinements may be attractive. For instance, XSD’s model-group definitions should be leveraged for providing more specific names instead of the generic nominations above.

There is an obvious problem with the nomination-based approach. The programmer must be intimately familiar with the (nesting) structure of the content model because it is resembled by local OO types and it therefore needs to be observed by member-based access. The resulting style of queries does not deliver on the expectation of an XPath-like child axis.

Relax nested content models so that they become flat

So far we tried to derive object types whose nesting resembled the nesting of the content model. Instead, we may trade precision for simplicity. One option is to relax the content models, in the formal sense of subset order on tree languages, so that the resulting content model is flat (and queries can be simpler).

For instance, we can relax a nested choice to a nested sequence over optional items; cf. the upper part of Fig. 29. This is essentially another interpretation of the mapping rule that we already discussed for choices; cf. Sec. 4.2. This relaxation would result in

<pre> plusOnSequence = (a b)+ <xs:element name="plusOnSequence"> <xs:complexType> <xs:sequence maxOccurs="unbounded"> <xs:element name="a" type="x"/> <xs:element name="b" type="y"/> </xs:sequence> </xs:complexType> </xs:element> </pre>
<p><u>Informal summary of nomination</u></p> <p>Before: plusOnSequence = (a b)⁺ After: plusOnSequence = Sequence⁺ where Sequence = a b</p>
<p><u>Resulting object model</u></p> <pre> public class plusOnSequence { public List<Sequence> AsList; public struct Sequence { public x a; public y b; } } </pre>

Fig. 28. ‘AsList’ access to repeating, nested sequence

a representation type that is too rich, if we were using plain objects over fields, but a property-based API with appropriate setter implementations may still enforce ‘mutual exclusion’ for the branches of the choice.

We face a more aggressive example of relaxation when nested, repeating composites are reduced to a repeating, element wildcard, say ‘any’. In the lower part of Fig. 29, we carry out such a relaxation for two examples. The drawbacks of these relaxations are obvious. First, there is a naming issue: How do we name the property that provides access to the relaxed collection? The ‘name concatenation’ approach used in the figure (cf. aAndB) is strikingly pragmatic, and does not scale for more complex composites. Second, we lose some degree of static typing for such relaxed composites.

Some extra static typing is obtainable, if we use a repeating choice over all elements in the content model. (This form of relaxation can be compared with the notion of prime-type conversion in the XQuery semantics [69].) When mapping such an ‘artificial’, repeating choice to an object model, we can use the different element names (say, types of elements) for specialized access to an otherwise untyped list.

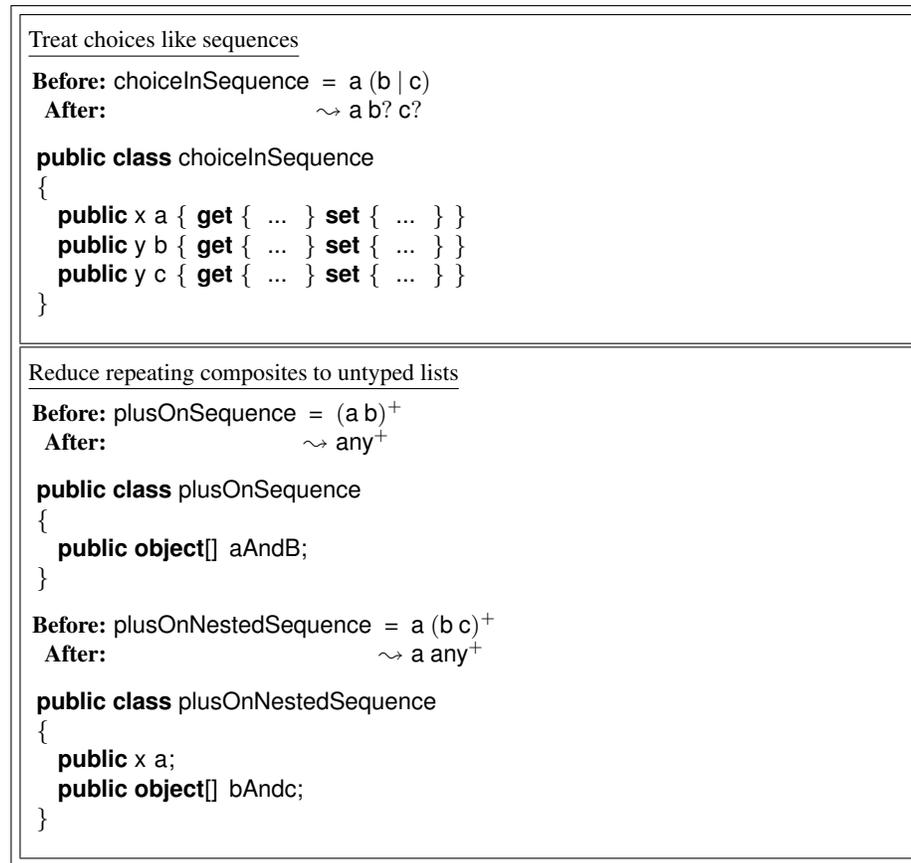


Fig. 29. Relax nested content models to become flat

View nested content models in a flat manner

For the sake of programming convenience, the schema-derived object types may ‘view’ the XML data as if it was of a very simple shape. Such views are particularly convenient for queries, but we need to be prepared to encounter update challenges. Views generally do not *relax* in the formal sense of subset order on tree languages. Here is a simple strategy for devising views on nested composites: relax choices as sequences, propagate quantifiers ($?$, $+$, $*$) over sequences to the components, and apply simplifications.

- Relax choices
 - $e_1 \mid \dots \mid e_n \rightsquigarrow e_1^? \dots e_n^?$
- Propagate quantifiers
 - $(e_1 \dots e_n)^q \rightsquigarrow e_1^q \dots e_n^q$
- Simplification rules
 - $e^{q^q} \rightsquigarrow e^q$
 - $e^{?^*} \rightsquigarrow e^*$

<pre> plusInAndOut = a (b⁺ c)⁺ <xs:element name="plusInAndOut"> <xs:complexType> <xs:sequence> <xs:element name="a" type="x"/> <xs:sequence maxOccurs="unbounded"> <xs:element name="b" type="y" maxOccurs="unbounded"/> <xs:element name="c" type="z"/> </xs:sequence> </xs:sequence> </xs:complexType> </xs:element> </pre>
<p>Mapping option: no support for round-tripping</p> <pre> public class plusInAndOut { public x a; public y[] b; public z[] c; } </pre>
<p>Mapping option: untyped container + typed getters</p> <pre> public class plusInAndOut { private enum label {a,b,c}; private List<Pair<label,object>> content; public x a { get { ... } set { ... } } public IEnumerable<y> b { get { ... } set { ... } } public IEnumerable<z> c { get { ... } set { ... } } } </pre>

Fig. 30. Views on content models

- $e^{?+} \rightsquigarrow e^*$
- etc.

The propagation rule clearly goes beyond relaxation. (Once we take recurrent element names into account, the rules become slightly more complicated.) Let us apply these rules to the most complex pattern in our suite — `plusInAndOut`:

Content model: `plusInAndOut = a (b+ c)+`
 Derived view: $\rightsquigarrow a b^+ c^+$

Fig. 30 shows two mapping options for the content model `plusInAndOut`. The class at the top is the result of simply mapping the derived view to a plain object type with

read/write-enabled fields. (The `xsd.exe` tool derives this class.) The grouping of `b`'s and `c`'s is not maintained in any way, thereby sacrificing round-tripping.

The class at the bottom of Fig. 30 devises an untyped (private) list to store all subtrees. For each element name, there is a getter and a setter, whose implementations actually leverage the fact that element names are stored. There are several challenges related to this approach. First, the mere definition of the correct semantics for the insertion mode of setters is non-trivial. Second, the programming model for constructing and updating nested composites is not straightforward, given that the original grouping for the nested composites is no longer discoverable. Third, normal type checking is insufficient to guarantee valid content at all times; hence, some scheme of dynamic checking may be necessary. Fourth, property access to a heterogeneous list is less efficient than field access for 'plain (sub)objects'.

4.4 Local elements

A number of OO languages support 'nested classes' of different kinds. However, nesting is hardly seen as a data-modeling idiom. Nested classes are instead used for advanced forms of encapsulation and programming with closures. The use of nested classes is typically discouraged if it only serves for a namespace-like purpose such as qualification. In contrast, the use of local element declarations is established at least for XSD.

Fig. 31 shows an XML schema for libraries (say, collections of books) and corresponding C# classes that were obtained by a nesting-preserving mapping. Unfortunately, nested classes are inconvenient when used for the construction of 'data objects' because the scoping rules of OO languages do not provide any shorthand for accessing nested classes. This weakness is illustrated in Fig. 31; the sample code constructs a single book instance and adds it to a given library `lib`; notice the qualified names `library.bookType` and `library.bookType.authorType`.

Hence, it may seem that XML schemas should generally be flattened prior to the actual X-to-O mapping. There are several problems with this approach. Most obviously, flattening bears the potential of causing clashes in the global scope. Also, the promotion of many locals to the global scope may make it more difficult to comprehend the data model. Furthermore, there is some element of arbitrariness in so far that there is more than just one flattening method, e.g.:

- Extract anonymous complex types of local elements as global type definitions.
- Promote local element declarations of complex types to the global scope.
- Promote local element declarations of simple types, too, for uniformity.
- Potentially even promote local attribute declarations, for uniformity.

The first two options are illustrated in Fig. 32. The last two options are phrased as refinements of the second option, and ultimately suggest 'universal promotion', which may appear as the most principled option; cf. Fig. 33. However, universal promotion, when applied to the library example, causes an 'ontological clash'. The original schema comprised two local element declarations with label `name`; they were unified in the result, which was possible because the declarations also agreed on the element type.

A nested element declaration

```

<xs:element name="library">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="book" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="author" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="title" type="xs:string"/>
                  <xs:element name="name" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="publisher" type="xs:string"/>
            <xs:element name="year" type="xs:gYear"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

A mapping option that preserves nesting

```

public class library
{
  public List<bookType> book;
  public class bookType
  {
    public string title ;
    public List<authorType> author;
    public string publisher;
    public System.DateTime year;
    public class authorType
    {
      public string title ;
      public string name;
    }
  }
}

```

Illustration of object construction

```

lib .book.Add(
  new library .bookType {
    title      = "COBOL Unleashed",
    author     = new List<library .bookType .authorType> {
      new library .bookType .authorType { name = "Jon Wessler" }
    },
    publisher  = "Macmillan Computer Publishing",
    year       = new System.DateTime(1998,9,1) } );

```

Fig. 31. Illustration of mapping for nested element declarations

Typed-oriented flattening ('Venetian blind')

```

<xs:element name="library">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="book" type="book" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="book">
  <xs:sequence>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="author" type="author" maxOccurs="unbounded"/>
    <xs:element name="publisher" type="xs:string"/>
    <xs:element name="year" type="xs:gYear"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="author">
  <xs:sequence>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="name" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

```

Element-oriented flattening ('Salami slice')

```

<xs:element name="library">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="book" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element ref="author" maxOccurs="unbounded"/>
      <xs:element name="publisher" type="xs:string"/>
      <xs:element name="year" type="xs:gYear"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

Fig. 32. Flattened schemas for Fig. 31 in two different styles

Reflection reveals that both kinds of names are different (in an ontological sense);¹⁹ hence, it is not acceptable to unify the declarations. To summarize, (universal) promotion cannot be performed, in general. This problem also vitally contributes to the overall

¹⁹ We face the title of a book vs. the title of a person. This example is adopted from a related mailing list discussion: <http://xsd.stylusstudio.com/2006Apr/post00002.htm>.

```

<xs:element name="library">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="book" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="title"/>
      <xs:element ref="author" maxOccurs="unbounded"/>
      <xs:element ref="publisher"/>
      <xs:element ref="year"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="title"/>
      <xs:element ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="title" type="xs:string"/>
<xs:element name="name" type="xs:string"/>
<xs:element name="publisher" type="xs:string"/>
<xs:element name="year" type="xs:gYear"/>

```

Fig. 33. Extremely flat style

difficulty to favor a node-label-preserving mapping because flattening may be needed in preparation of such a mapping, if we agree that nested object types are inconvenient, as discussed above. This analysis provides further evidence for the X/O impedance mismatch.

4.5 Element templates

When using object models for data representation, we mainly use one form of type abstraction for data: *classes* with nominal type equality and explicitly declared inheritance (subtyping) relationships. The situation for data modeling with XSD is less homogeneous. There are different kinds of types. XSD's *element declarations* qualify as *types* in the common sense of programming-language theory. That is, element declarations (both global and local) denote sets of semantically *meaningful* values: certain sets of XML trees. XSD's (complex) *type definitions* are not exactly like types in programming language theory. They seem to be modeling 'unlabeled trees'; such values do not really exist. We may think of (complex) type definitions as incomplete element declarations, say element templates.

One may feel tempted to dismiss such templates as macros for structural types, but this would be an oversimplification. For instance, the semantically meaningful notion of element substitution ('inheritance') is tied to type substitution; cf. Sec. 4.6 and Sec. 4.7.

For the record, XSD also comprises attribute-group definitions and model-group definitions, which we neglect here. Even if we focus on just element declarations and

complex-type definitions, one may wonder whether these different forms of type abstractions must be regarded as an XSD idiosyncrasy. Arguably, there is a fundamental need for element templates because they enable an important form of reuse. To provide another data point, RELAX NG offers so-called ‘named patterns’ that are akin to the complex-type definitions of XSD.

Fig. 34 shows a typical XSD sample that illustrates the reuse value of complex-type definitions. The address type is used in two positions: billing addresses and shipping addresses. The exclusive use of element declarations would certainly simplify the devising of X-to-O mappings, but the example suggests that an elimination of complex types is generally not feasible without prohibitive code duplication.

Fig. 34 also shows the ‘most obvious’ option for mapping the customer schema. Both, global element declarations and complex-type definitions are mapped to classes that look alike, which is troublesome for the following reason. One might create an `Address` instance and then expect to be able to serialize it. However, the class `Address` is not associated with any element tag. Should we disallow serialization? Should we instead serialize addresses in a special way, as a sort of XML ‘fragment’? Should we instead use the type name as the element name? How does the OO programmer comprehend the technical difference between the different kinds of object types: `Address` vs. `Customer`?

Also, the mapping further illustrates the dubiety of the common, node-label-omitting mapping. To see this, consider the following code fragment that operates on the object model of Fig. 34:

```
var myAddr = new Address();
myAddr.name = "Fred Mueller";
myAddr.street = "88th NE CT. place";
var myCust = new customer();
myCust.id = "123456";
myCust.billingAddress = myAddr;
myCust.shippingAddress = myCust.billingAddress; // What's this ?
```

That is, we assign a billing address to a field for a shipping address. In the original value domains for the XML addresses, this assignment does not make sense because we are facing different element names — `billingAddress` vs. `shippingAddress`, i.e., we face different types in the sense of programming-language theory. Hence, the above code seems to require that the element name `billingAddress` is replaced by `shippingAddress` underneath. It is quite unusual to think of an assignment that involves a hidden type change.

There is perhaps one way to improve on the indistinguishable mapping of global element declarations and complex-type definitions. We may ‘de-prioritize’ complex types and map them to *interfaces* as opposed to classes. These interfaces may be instrumental in establishing useful type-level relationships (rather than plain assignability). Here is an interface for the complex type `Address`:

```
public interface Address
{
    string name { get; set; }
    string street { get; set; }
}
```

An XML schema that leverages complex types for reuse

```

<xs:element name="customer">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="billingAddress" type="Address"/>
      <xs:element name="shippingAddress" type="Address"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="Address">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="street" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

```

An illustrative XML instance

```

<customer>
  <id>123456</id>
  <billingAddress>
    <name>Fred Mueller</name>
    <street>88th NE CT. place</street>
  </billingAddress>
  <shippingAddress>
    <name>Fred Mueller</name>
    <street>88th NE CT. place</street>
  </shippingAddress>
</customer>

```

An obvious mapping option

```

public class customer {
  public string id ;
  public Address billingAddress;
  public Address shippingAddress;
}
public class Address {
  public string name;
  public string street ;
}

```

Fig. 34. Illustration of the reuse aspect for complex types

Let us illustrate the interface-based scheme. To this end, we assume a node-label-preserving mapping. There are classes `billingAddress` and `shippingAddress`, which implement the interface `Address`. Further, these classes provide a *copy constructor* using an `Address`-bounded argument. As a result, the earlier code fragment can be rewritten as follows:²⁰

```
var myAddr = new billingAddress();
myAddr.name = "Fred Mueller";
myAddr.street = "88th NE CT. place";
var myCust = new customer();
myCust.id = "123456";
myCust.billingAddress = myAddr;
myCust.shippingAddress = new shippingAddress(myAddr);
```

Hence, we can reuse a given address (say a billing address) in creating another address (say a shipping address). The use of the copy constructor makes explicit the conceptual type change that was previously hidden. The approach is not general: (i) it requires the (uncommon) node-label-preserving mapping; (ii) it cannot be used in combination with type derivation, as we will discuss in the next subsection. This failure of an attempted conceptual clean-up provides further evidence for the X/O impedance mismatch.

4.6 Type extension

XSD's type derivation by extension is (intentionally) similar to OO subclassing. That is, one can define new types by essentially extending other types (referred to by name). Extended types are substitutable for base types; cf. Fig. 35 for an example.²¹ To provide another data point, RELAX NG does not provide such linguistic support, but one can still use a combination of 'design patterns' and annotations to effectively model the same kind of relationship [59,60].

The interface-oriented mapping rule of the previous section falls short for types that engage in type derivation (except for a special case that we will identify in Sec. 4.7). Classes are needed for derived complex types and their base types because we must be able to construct objects of all the types corresponding to the different members in the derivation hierarchy. For instance, in the example, there is just one root element, `product`, which can be of two different types, `ProductType` and `ShirtType`.

Hence we must be prepared to designate classes both to element declarations and complex-type definitions. A sensible optimization comes to mind. We may attempt to omit classes for element declarations with a nominal element type, namely, when the nominal element type implies the element name and all derived types imply the same element name. Consider again the schema in Fig. 35: `product` elements are of type `ProductType` or `ShirtType`, and there are no other elements of these types. Hence no designated class is needed for the root `product`. (Indeed, the `xsd.exe` tool maps the schema in this manner.)

²⁰ For the record, the shown code rests on the assumption that implicit casts are available for all element declarations with simple element types. Otherwise, extra 'element constructors' would be needed in the various assignments such as `new name("Fred Mueller")` instead of `"Fred Mueller"`.

²¹ A note on XSD: Notice the attribute `xsi:type="ShirtType"` in the sample instance. In this manner, validation is informed to expect the derived type in place of the base type.

An XML schema that involves type derivation

```

<xs:element name="items">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="product" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="product" type="ProductType"/>
<xs:complexType name="ProductType">
  <xs:sequence>
    <xs:element name="number" type="xs:int"/>
    <xs:element name="name" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ShirtType">
  <xs:complexContent>
    <xs:extension base="ProductType">
      <xs:sequence>
        <xs:element name="size" type="ShirtSizeType"/> <!-- ShirtSizeType elided -->
        <xs:element name="color" type="ColorType"/> <!-- ColorType elided -->
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

```

A sample instance exercising type substitution

```

<items>
  <product xsi:type="ShirtType">
    <number>557</number>
    <name>Short-Sleeved Linen Blouse</name>
    <size>10</size>
    <color value="blue"/>
  </product>
</items>

```

An obvious mapping option

```

public class items
{
  public List<ProductType> product;
}
public class ProductType
{
  public int number;
  public string name;
}
public class ShirtType : ProductType
{
  public ShirtSizeType size;
  public ColorType color;
}

```

Fig. 35. Illustration of mapping type derivation by extension

```

public class product : XLabelled
{
    public product() { content=new ProductType(); }

    public product(ProductType Content) { content=Content; }

    public ProductType Content { get { return content; }}

    public int number {
        get { return Content.number; }
        set { Content.number = value; }
    }
    public string name {
        get { return Content.name; }
        set { Content.name = value; }
    }
    private ProductType content;
}

```

Fig. 36. Mapping roots with nominal element types

We cannot apply this optimization, when there are multiple (global) element declarations referring to the same element type (or to types related to each other by type derivation). Hence, we have to accept ultimately that both element declarations and complex-type definitions are mapped to classes — except for some special cases, *if* we are willing to cater for exceptions. As a last attempt of restoring some discipline (that helps avoiding confusions between element declarations and complex-type definitions), we may assume two different base types for schema-derived classes, thereby replacing the implicit base class `object`, used until now:

- `XLabelled` — Base class for element declarations.
- `XUnlabelled` — Base class for complex-type definitions.

This discipline provides a bit of discoverability. Also, we may assume that only subclasses of `XLabelled` implement the general de-/serialization protocol, thereby avoiding accidental uses of complex types for serialization.

Global element declarations with nominal element types are mapped according to a ‘wrapper scheme’ that reflects the fact that the wrapper only provides the element name while the wrappee provides all the actual ‘structure’, i.e., children and attributes; cf. Fig 36 for an illustration. The constructor of the wrapper class takes a wrappee. The interface of the base type of the wrappee is re-implemented by the wrapper type (through forwarding).

Let us investigate the convenience and clarity of the resulting programming model. To this end, we revisit the example for populating two addresses (billing vs. shipping) based on elements of the same nominal type for addresses. In the following code fragment, we attempt to reuse an address object:

```

var myAddr = new Address();
myAddr.name = "Fred Mueller";
myAddr.street = "88th NE CT. place";
var myCust = new customer();
myCust.id = "123456";
myCust.billingAddress = new billingAddress(myAddr);
myCust.shippingAddress = new shippingAddress(myAddr);

```

Hence, we pass the ‘unlabeled object’ `myAddr` to the wrapping constructors for the labeled types of billing and shipping addresses. One may wonder what exactly the semantics of the wrapping constructors should be. We recall that the interface-based mapping of the previous subsection assumed a *copy constructor* taking a labeled object and creating a labeled object. A plain copy semantics does not seem to be appropriate for the wrapping constructors because this would imply that ‘unlabeled objects’ are never integrated into object graphs, as is. A ‘no-op’ semantics, i.e., wrapping ‘unlabeled objects’ any number of times, does not seem to be appropriate either because it could never meet the tree invariant for XML. An alternative semantics would then be to *‘parent’ unlabeled objects when they are used for the first time, and to copy (clone) them from there on*. This discussion leaves a bad aftertaste, and hence, provides further evidence for the X/O impedance mismatch.

4.7 Element substitution

In XSD, element declarations can engage in so-called substitution groups, thereby providing a form of substitutability for elements. (RELAX NG would again leverage ‘design patterns’ to this end [59,60].) As a simple illustration of substitution groups, let us add an element declaration to the earlier schema for products in Fig. 35:

```
<xs:element name="shirt" type="ShirtType" substitutionGroup="product"/>
```

As a result, there are the following associations:

Element name	Nominal element type
product	ProductType
shirt	ShirtType

According to the rules of XSD, all members of a substitution group, except for leaves, must be of a nominal element type. In practice, this implies that every ‘concept’ (such as the concept of ‘shirts’) gives rise to *two* schema abstractions; cf. the element declaration `shirt` and the complex-type definition `ShirtType`. One may expect a form of element substitution that does not rely on nominal element types.

There are the following mapping options for substitution groups:

- *The wrapping option*: Based on the discussion in Sec. 4.6, we map the members of the substitution group (which are element declarations with a nominal element type) to wrapper types on the object types for the element types. Both the wrapper and the wrappee classes engage in class inheritance.

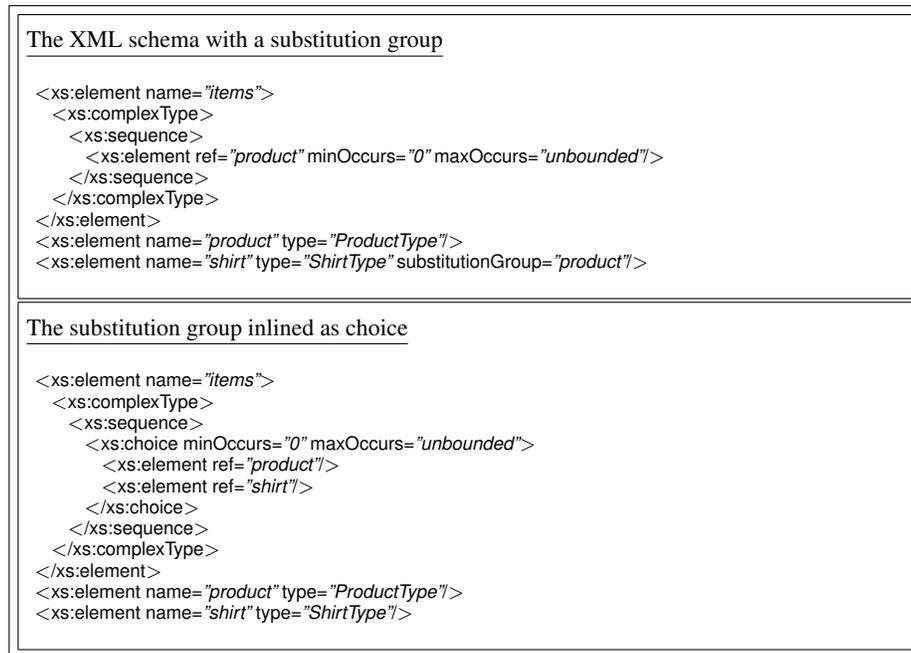


Fig. 37. Map substitution groups as choices

- *The choice option:* We inline each reference to the head of a substitution group as a choice over all of its members; cf. Fig. 37. Thereby, we effectively eliminate the use of substitution groups. The members of a substitution group are still to be mapped to wrapper classes, but subclassing is restricted to unlabeled object types.
- *The normalization option:* If the element types of substitution-group members are not used elsewhere in the schema, and if there is one member per possible type (in the type-derivation hierarchy), then we can potentially omit the object types for the type-derivation hierarchy and map the substitution groups to a plain class hierarchy; cf. Fig. 38 for an illustration.

With the first option, the OO programmer may ‘get lost’ in the two parallel class hierarchies. With the second option, the OO programmer may be overwhelmed by extra-large choices. Both options transport the ‘doubled’ number of schema abstractions to the object model, thereby becoming unpalatable. The third option would lead to a concise object model, but its applicability is subject to preconditions. Also it takes away (subtle) options for content construction, as illustrated by the following sample data:

```

<items>
  <product xsi:type="ShirtType"> ... content elided ... </product>
  <shirt> ... content elided ... </shirt>
</items>

```

<p>Mapping option: prefer element declarations over type definitions</p> <pre> public class product { public int number; public string name; } public class shirt : product { public ShirtSizeType size; public ColorType color; } </pre>
<p>Mapping option: prefer type definitions over element declarations</p> <pre> public class ProductType { public int number; public string name; } public class ShirtType : ProductType { public ShirtSizeType size; public ColorType color; } </pre>

Fig. 38. Illustration of mapping for substitution groups

That is, the collection comprises two shirts; the first one (`<product xsi:type="ShirtType"> ...`) exploits type substitution; the second one (`<shirt> ...`) exploits element substitution. The normalized object model of Fig. 38 cannot easily differentiate these two kinds of shirts along construction and observation.

Further, the normalization option is challenged by a naming issue: should we prioritize element names over type names, or vice versa? We would like to adopt the element names as class names (because element declarations may count as the primary form of XML types). However, we must adopt the type names as class names, as soon as the type-derivation hierarchy makes more type distinctions than the associated substitution group for elements. Likewise, we must adopt the element names, as soon as the substitution group makes more type distinctions. Hence, a relatively small change (such as the addition of one global schema component) may trigger a different naming scheme for the entire schema.

4.8 Type restriction

XSD provides another form of type derivation — by restriction. We will focus here on restriction for complex types, and defer the coverage of simple types to the next subsec-

```

<xs:complexType name="foo">
  <xs:sequence>
    <xs:element name="x" type="foo" minOccurs="0"/>
    <xs:element name="y" type="foo"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="bar">
  <xs:complexContent>
    <xs:restriction base="foo">
      <xs:sequence>
        <!-- Optional "x" dropped out. -->
        <xs:element name="y" type="foo"/>
      </xs:sequence>
    </xs:restriction >
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="abc">
  <xs:complexContent>
    <xs:restriction base="bar">
      <xs:sequence>
        <!-- Require subtype for "y". -->
        <xs:element name="y" type="bar"/>
      </xs:sequence>
    </xs:restriction >
  </xs:complexContent>
</xs:complexType>

```

Fig. 39. Illustration of complex-type derivation by restriction

tion. Restricted types may be used in type substitution (`'xsi:type'`) in the same manner as extended types. The typical OO language has no counterpart for type derivation by restriction. In an effort to at least preserve XSD's substitutability, we suggest to map restriction relationships again to class inheritance, just as we did for extension relationships. It remains to further justify this mapping rule.

Fig. 39 shows a contrived example that demonstrates principled forms of complex-type restriction by a derivation chain `foo`, `bar` and `abc`. In the derivation of `bar` from `foo`, *an optional particle is eliminated*. In the derivation of `abc` from `bar`, *the element type of a particle is restricted to a subtype*.

There is the following scale for mapping restricted types to classes:

1. Just inherit the base class without any change; neglect restrictions.
2. Enforce restrictions when validation is requested for an object.
3. Enforce some restrictions as invariants based on dynamic checks.
4. Enforce some restrictions by means of the static type system.

Fig. 40 shows a mapping option where the restrictions are enforced by dynamically checked setters. (Depending on mapping rules and schema patterns, dynamic checks

```

public class foo
{
  public virtual foo x { get { return ._x; } set { ._x = value; } }
  public virtual foo y { get { return ._y; } set { ._y = value; } }
  private foo ._x;
  private foo ._y;
}
public class bar : foo
{
  public override foo x { set { Trace.Assert(value==null); base.x = value; } }
}
public class abc : bar
{
  public override foo y { set { Trace.Assert(value is bar); base.y = value; } }
}

```

Fig. 40. A mapping option for complex-type derivation by restriction

may also be required for getters.) Suppose we would want to enforce the restrictions by means of the static type system. In the derivation of *bar*, we would need to *remove* a member. Member removal is unsupported by several mainstream OO languages (including C#), and it is a controversial capability anyhow. In the derivation of *abc*, we would need to *co-variantly modify* the argument type of a setter. Covariance for arguments is unsupported by most mainstream OO languages (including C#), and it is a controversial capability because of the difficulty to reconcile static type safety.

The use of class inheritance for restriction relationships may be said to systematically violate the substitution principle [36]. The problem is that subclass setters are constrained by stronger preconditions than the base-class setters. This situation is reminiscent of ‘cheating servers’ [43], which are in conflict with behavior-preserving subtyping and design-by-contract (DBC). We may adopt a mitigation technique that is also offered in the DBC literature. That is, we may take the view that restriction conditions are instance parameters in a sense. To this end, the object model anticipates ‘variable’ restriction conditions in separate, virtual, Boolean methods — one per property. A restricted subclass may then override the restriction condition. Fig. 41 illustrates this ‘trick’ for the running example.

4.9 Simple types

Up to now, we used built-in simple types, which we mapped to a fixed set of primitive programming-language types such as *string* and *double* for C#. XSD also provides list types and union types, which we neglect here for brevity. Further, XSD provides type derivation by restriction for simple types, which we will discuss now.

Fig. 42 illustrates several mapping options for simple types when type derivation by restriction is involved. The first option assumes that all simple types (including the derived ones) are mapped to primitive types of the programming language at hand. In

```

public class foo
{
    // Boolean conditions for virtual contracts
    protected virtual bool xValid(foo x) { return true; }
    protected virtual bool yValid(foo y) { return true; }

    // State access with dynamically checked properties
    public virtual foo x {
        get { Trace.Assert(xValid(_x)); return _x; }
        set { Trace.Assert(xValid(value)); _x = value; }
    }
    public virtual foo y {
        get { Trace.Assert(yValid(_y)); return _y; }
        set { Trace.Assert(yValid(value)); _y = value; }
    }
    private foo _x;
    private foo _y;
}

public class bar : foo
{
    protected override bool xValid(foo x) {
        return base.xValid(x) && x == null;
    }
}

public class abc : bar
{
    protected override bool yValid(foo y) {
        return base.yValid(y) && y is bar;
    }
}

```

Fig. 41. More DBC-compliant, restriction-ready object types

fact, a restricted type is mapped to the same type as its base type. In the sample schema in the figure, there is the restricted type `uint42` with `xs:unsignedInt` as its base type. Both are mapped to C#'s type `uint`.

As a first enhancement, we may transport restrictions as dynamic checks into the object model; cf. the second mapping option in Fig. 42. The use of dynamic checks is similar to the treatment of `minOccurs`/`maxOccurs` constraints and complex-type restrictions, as discussed earlier. In particular, setters are constrained by preconditions that model the schema restrictions.

As a second enhancement, we may designate object types (or struct types) to nominally defined simple types. Such a mapping option is motivated by the insight that even simple types may serve an important role in program comprehension and software evolution [19]. Also, the use of designated types would enable static type checking

An XML schema with a simple-type restriction

```

<xs:simpleType name="uint42">
  <xs:restriction base="xs:unsignedInt">
    <xs:maxInclusive value="42"/>
  </xs:restriction >
</xs:simpleType>
<xs:element name="simples">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="anUInt" type="xs:unsignedInt"/> <!-- unrestricted type -->
      <xs:element name="anUInt42" type="uint42"/> <!-- restricted type -->
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

Mapping option: replace derived types by their base types

```

public class simples
{
  public uint anUInt;
  public uint anUInt42;
}

```

Mapping option: enforce restrictions by dynamic checks

```

public class simples
{
  public uint anUInt {
    get { return ._anUInt; }
    set { ._anUInt = value; } }
  public uint anUInt42 {
    get { return ._anUInt42; }
    set { Trace.Assert(value<=42); ._anUInt42 = value; } }
  private uint ._anUInt;
  private uint ._anUInt42;
}

```

Mapping option: structs designated to simple-type definitions

```

public struct uint42
{
  public static implicit operator uint42(uint it) {
    Trace.Assert(it<=42); return new uint42 { value = it };
  }
  public static implicit operator uint(uint42 it) {
    return it .value;
  }
  private uint value;
}
public class simples
{
  public uint anUInt;
  public uint42 anUInt42;
}

```

Fig. 42. Lossy mappings for simple types

```

Schema-derived classes

public class uint42 : XsUnsignedInt
{
    public static implicit operator uint42(uint it) {
        Trace.Assert(it <= 42);
        return new uint42 { value = it };
    }
    public static implicit operator uint(uint42 it) {
        return (uint) it .value;
    }
}

public class simples
{
    public XsUnsignedInt anUInt;
    public uint42 anUInt42;
}

Predefined classes for XSD's built-in simple types

// Base class of all simple types
public abstract class XsAnySimpleType
{
    protected object value;
}

// The counterpart for xs:unsignedInt
public class XsUnsignedInt : XsUnsignedLong
{
    public static implicit operator XsUnsignedInt(uint it) {
        return new XsUnsignedInt { value = it };
    }
    public static implicit operator uint(XsUnsignedInt it) {
        return (uint) it .value;
    }
}

// Other simple types, likewise
public class XsUnsignedLong : XsNonNegativeInteger { ... }
public class XsNonNegativeInteger : XsInteger { ... }
public class XsInteger : XsDecimal { ... }
public class XsDecimal : XsAnySimpleType { ... }

```

Fig. 43. Mapping simple types to a designated class hierarchy

to guarantee that a value of a given restricted type does indeed meet the restrictions, thereby making repeated dynamic checks unnecessary.

The use of struct types (as opposed to classes) may be particularly attractive because struct types are potentially exempt from charges (subject to compiler optimizations) — if they only wrap a single component. The structs-based mapping option is illustrated at the bottom of Fig. 42. We use implicit casts for both directions of mediating between the struct type for the simple type and the associated primitive type. The restriction is only checked when a value of the restricted type is constructed. Unfortunately, a structs-based mapping does not enable proper substitution for simple types (because there is no subtyping for struct types). Consider the following XML data, which exercises simple-type substitution for the first child element labeled `anUInt`:

```
<simples xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <anUInt xsi:type="uint42">41</anUInt>
  <anUInt42>41</anUInt42>
</simples>
```

The property for the `anUInt` element is of type `uint` (or perhaps of a designated struct type for `xs:unsignedInt`, if we decide to designate struct types for all of XSD's built-in simple types). Hence, the getter (based on a structs-based mapping) would never report the status of the datum to be of the restricted type `uint42`.

Fig. 43 makes the next step by designating wrapper classes to simple types. There are also predefined classes for XSD's built-in simple types so that XSD's substitutability can be preserved completely. Clearly, this approach is quite costly because of 'boxing' (cf. the field of type `object` in `XsAnySimpleType`) and wrapping (due to the use of classes). Also, the programming convenience is arguably impaired because OO programmers may prefer object models that leverage the familiar primitive types of their OO language (as opposed to the XSD types). One may provide implicit casts so that the objects of a 'simple-type class' can be used whenever the associated primitive type is expected, but the resulting hybrid may be difficult to comprehend by the programmer. To summarize, there is no fully satisfactory simple-type mapping. This circumstance also provides further evidence for the X/O impedance mismatch.

5 Concluding remarks

The X/O impedance mismatch has to be primarily attributed to fundamental differences in the data models for XML and objects. The data models are so much different that the expectation of using 'plain objects' for the typed representation of XML trees should be internationally condemned. It appears that the X/O impedance mismatch must also be attributed to differences in the type systems (the type languages) for XML and objects. However, it is important to note that the differences in type systems are largely implied by the underlying data models. Finally, the complexity of XSD and its freewheeling use in the wild add considerably to the severity of the X/O impedance mismatch.

The work reported in this paper clearly substantiates that a full resolution of the X/O impedance mismatch is fundamentally impossible. The optimistic (more pragmatic) interpretation of our work is that the understanding of (canonical) X-to-O mappings has

matured, and hence, more ambitious X-to-O mappings are in reach such that XML semantics and schema constraints are better preserved by the resulting object models, while also hiding the complexities and idiosyncrasies of XML and XSD — to some extent. More ambitious X-to-O mappings buy us time until something simpler than the current standards (XML 1.0/1.1+XSD 1.0/1.1) emerges. The pessimistic (more intellectual) interpretation of our work is that (canonical) X-to-O mappings remain hacks (and XSD is by far too complicated), no matter what, and hence a famous comment by Dijkstra (originally advised for Cobol) comes to mind [16]: “*you can really do only one of two things: fight the disease or pretend that it does not exist*”.

Acknowledgments The authors acknowledge interactions with Umut Alev, Brian Beckmann, Sergey Dubinets, Priya Lakshminarayanan, Chris Lovett, Sergey Melnik, Dave Remy, Dan Rogers, Mark Shields, Huseyin Ulger, and Eugene Veselov. More generally, the authors have benefited from countless discussions with members of the Data Programmability team and further individuals at Microsoft.

References

1. S. Amer-Yahia, F. Du, and J. Freire. A comprehensive solution to the XML-to-relational mapping problem. In A. H. F. Laender, D. Lee, and M. Ronthaler, editors, *6th ACM CIKM International Workshop on Web Information and Data Management (WIDM'04), Proceedings*, pages 31–38. ACM Press, 2004.
2. F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A Type-Preserving XML Schema-Haskell Data Binding. In B. Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium (PADL'04), Proceedings*, volume 3057 of *LNCS*, pages 71–85. Springer-Verlag, 2004.
3. A. I. Baars, A. Löh, and S. D. Swierstra. Parsing permutation phrases. *Journal of Functional Programming*, 14(6):635–646, 2004.
4. F. Bancilhon and N. Spyrtos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.
5. D. Bau. The Design of XMLBeans, 2003. Parts 1–3: http://davidbau.com/archives/2003/11/14/the_design_of_xmlbeans_part_1.html; http://davidbau.com/archives/2003/11/19/the_design_of_xmlbeans_part_2.html; http://davidbau.com/archives/2003/12/18/the_design_of_xmlbeans_part_3.html.
6. G. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in $C\omega$. In A. P. Black, editor, *ECOOP'05, Object-Oriented Programming, 19th European Conference, Proceedings*, volume 3586 of *LNCS*, pages 287–311. Springer-Verlag, 2005.
7. P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Bridging the XML Relational Divide with LegoDB. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *19th International Conference on Data Engineering (ICDE'03), Proceedings*, pages 759–760. IEEE Computer Society, 2003.
8. P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *18th International Conference on Data Engineering (ICDE'02), Proceedings*, page 64. IEEE Computer Society, 2002.
9. R. Bordawekar, M. G. Burke, I. Peshansky, and M. Raghavachari. XJ: Integration of XML Processing into Java. In Castagna and Raghavachari [14], page 85.

10. R. Bourret. Mapping W3C Schemas to Object Schemas, Mar. 2001. <http://www.rpbouret.com/xml/SchemaMap.htm>.
11. R. Bourret. XML Data Binding Resources, 2007. Web site; <http://www.rpbouret.com/xml/XMLDataBinding.htm>.
12. F. Burton and R. Cameron. Pattern Matching with Abstract Data Types. *Journal of Functional Programming*, 3(2):171–190, 1993.
13. R. D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Programming Language Systems*, 2(1-4):85–94, 1993.
14. G. Castagna and M. Raghavachari, editors. *PLAN-X 2006 Informal Proceedings, Charleston, South Carolina, January 14, 2006*. BRICS, Department of Computer Science, University of Aarhus, 2006.
15. R. Crocker and G. L. Steele Jr., editors. *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'03*. ACM Press, 2003.
16. E. W. Dijkstra. EWD 498: How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective*, pages 129–131. Springer-Verlag, 1992.
17. P. Downey. W3C XML Schema Patterns for Databinding. In *Conference Proceedings XML'06*, Dec. 2006.
18. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In Crocker and L. Steele Jr. [15], pages 302–312.
19. M. Fowler. When to Make a Type. *IEEE Software*, pages 12–13, Jan./Feb. 2003.
20. V. Gapeyev, F. Garillot, and B. C. Pierce. Statically Typed Document Transformation: An Xtatic Experience. In Castagna and Raghavachari [14], pages 2–13.
21. V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML Goes Native: Run-Time Representations for Xtatic. In R. Bodík, editor, *Compiler Construction, 14th International Conference, CC'05, Proceedings*, volume 3443 of LNCS, pages 43–58. Springer-Verlag, 2005.
22. V. Gapeyev and B. C. Pierce. Regular Object Types. In L. Cardelli, editor, *ECOOP'03, Object-Oriented Programming, 17th European Conference, Proceedings*, volume 2743 of LNCS, pages 151–175. Springer-Verlag, 2003.
23. R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In Crocker and L. Steele Jr. [15], pages 115–134.
24. G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.
25. M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: facilitating XML processing in Java. In A. Ellis and T. Hagino, editors, *WWW'05, 14th International Conference on World Wide Web, Proceedings*, pages 278–287. ACM Press, 2005.
26. M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, V. Sarkar, and R. Bordawekar. XJ: integration of XML processing into java. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *WWW'04, 13th International Conference on World Wide Web, Proceedings*, pages 340–341. ACM Press, 2004.
27. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.
28. IBM Research. XJ, 2005. <http://www.research.ibm.com/xj/>.
29. jsonJSON.org. Introducing JSON, 2006. Web site, <http://www.json.org/>.
30. C. Kirkegaard and A. Møller. Type Checking with XML Schema in XACT. In Castagna and Raghavachari [14], pages 14–23.

31. M. G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, and M. Mercaldi. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In *WWW'06, 15th International Conference on World Wide Web, Proceedings*, pages 93–102. ACM Press, 2006.
32. R. Lämmel, S. Kitsis, and D. Remy. Analysis of XML schema usage. In *Conference Proceedings XML'05*, Nov. 2005.
33. R. Lämmel and D. Remy. Functional OO Programming with Triangular Circles. In *Conference Proceedings XML'06*, Dec. 2006.
34. R. Lämmel, E. Visser, and J. Visser. Strategic Programming Meets Adaptive Programming. In *AOSD'03, 2nd International Conference on Aspect-Oriented Software Development, Proceedings*, pages 168–177. ACM Press, 2003.
35. K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
36. B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA'87, Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 17–34. ACM Press, 1987.
37. S. Loughran and E. Smith. Rethinking the Java SOAP Stack. Technical Report HPL-2005-83, 20050517, External, HP Labs, 2005.
38. B. McLaughlin. *Java and XML data binding*. Nutshell handbook. O'Reilly & Associates, Inc., 2002.
39. E. Meijer and B. Beckman. XLINQ: XML Programming Refactored (The Return Of The Monoids). In *Conference Proceedings XML'05*, Nov. 2005. See [46] for the LINQ portal. Xlinq is now called LINQ to XML.
40. E. Meijer and B. Beckman. XML Support in Visual Basic 9. In Castagna and Raghavachari [14], page 86.
41. E. Meijer, W. Schulte, and G. Bierman. Programming with Circles, Triangles and Rectangles. In *Conference Proceedings XML'03*, Dec. 2003.
42. E. Meijer, W. Schulte, and G. Bierman. Unifying Tables, Objects and Documents. In *Proceedings of Declarative Programming in the Context of OO Languages (DP-COOL)*, Sept. 2003.
43. B. Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, 1997.
44. Microsoft Corp. LINQ to XML Overview, 2005–2007. See [46] for the LINQ portal. LINQ to XML was formerly called Xlinq.
45. Microsoft Corp. C# Version 3.0 Specification, 2006–2007. See [46] for the LINQ portal.
46. Microsoft Corp. The LINQ Project, 2006–2007. <http://msdn.microsoft.com/netframework/future/linq/>.
47. Microsoft Corp. Windows Communication Foundation, 2006–2007. <http://windowscommunication.net/>.
48. G. S. Novak Jr. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, 1995.
49. OASIS. RELAX NG Specification, Dec. 2001. <http://www.oasis-open.org/committees/relax-ng/spec-20011203.html>.
50. M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, pages 671–681. ACM Press, 2005.
51. E. Perkins, M. Matsa, M. G. Kostoulas, A. Heifets, and N. Mendelsohn. Generation of efficient parsers through direct compilation of XML Schema grammars. *IBM Systems Journal*, 45(2):225–244, 2006.

52. M. Pradhan. Default mapping for annotated XML schema. *IBM developerWorks*, Apr. 2006. <http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0604pradhan2/>.
53. M. Shields and E. Meijer. Type-indexed rows. In *POPL'01: 28th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, Proceedings*, pages 261–275. ACM Press, 2001.
54. Sun Microsystems. The Java architecture for XML binding (JAXB), 2006. <http://java.sun.com/webservices/jaxb/>.
55. The Apache XML Project. XMLBeans, 2006. <http://xmlbeans.apache.org/>.
56. P. Thiemann. Modeling HTML in Haskell. In E. Pontelli and V. S. Costa, editors, *PADL'00, Practical Aspects of Declarative Languages, Proceedings*, volume 1753 of *LNCS*, pages 263–277. Springer-Verlag, 2000.
57. P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, 2002.
58. D. Thomas. The Impedance Imperative: Tuples + Objects + Infosets = Too Much Stuff! *Journal of Object Technology*, 2(5):7–12, Sept.–Oct. 2003.
59. E. van der Vlist. *RELAX NG*. O'Reilly, Dec. 2004.
60. E. van der Vlist. RELAX NG and W3C XML Schema compared (continued), July 2006. Weblog [http://eric.van-der-vlist.com/blog/2814_RELAX_NG_and_W3C_XML_Schema_compared_\(continued\).item](http://eric.van-der-vlist.com/blog/2814_RELAX_NG_and_W3C_XML_Schema_compared_(continued).item).
61. R. van Engelen, M. Govindaraju, , and W. Zhang. Exploring Remote Object Coherence in XML Web Services. In *ICWS'06, International Conference on WebServices, Proceedings*, pages 249–256. IEEE Computer Society, 2006.
62. S. Vinoski. RPC Under Fire. *IEEE Internet Computing*, 9(5):93–95, Sept./Oct. 2005.
63. J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices, OOPSLA 2001 Conference Proceedings*, 36(11):270–282, Nov. 2001.
64. W3C. Document Object Model (DOM), 1997–2003. <http://www.w3.org/DOM/>.
65. W3C. XML Path Language (XPath), Version 1.0, W3C Recommendation, Nov. 16 1999. <http://www.w3.org/TR/xpath>.
66. W3C. XML Information Set (Second Edition), 1999–2004. <http://www.w3.org/TR/xml-infoset/>.
67. W3C. XML Schema, 2000–2003. <http://www.w3.org/XML/Schema>.
68. W3C. Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation, Feb. 2004. <http://www.w3.org/TR/2004/REC-xml-20040204/>.
69. W3C. XQuery 1.0 and XPath 2.0 Formal Semantics, W3C Candidate Recommendation, 8 June 2006. <http://www.w3.org/TR/xquery-semantics/>.
70. P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL'87, 14th ACM SIGACT-SIGPLAN Symposium on Principles Of Programming Languages, Proceedings*, pages 307–313. ACM Press, 1987.
71. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices, Conference Proceedings of International Conference on Functional Programming (ICFP'99)*, 34(9):148–159, Sept. 1999.

A Extreme mapping options

This appendix illustrates some mapping options that work hard to convey schema constraints into the derived object models. Arguably, the options are too complicated for practical use.

A.1 Type-driven member access

Sec. 3.7 clarified that element references do not faithfully correspond to OO member names, and concluded that a type-driven access protocol may be more appropriate. In fact, we may use the types of children as (ambiguous) selectors, thereby mimicking the XPath model. To get started, we assume a mapping as follows:

- Map XSD namespaces to .NET namespaces in a 1:1 manner.²²
- Map element declarations to classes in a 1:1 manner.
- Use a node-label-preserving mapping; cf. Sec. 3.5.

The last item is important because it guarantees that the resulting object types preserve all type distinctions (read as ‘element-label distinctions’) that are prescribed by the original schema. That is, the object types are sufficient for type-driven access.

A type-driven replacement of “.” requires access operations for ‘getting’ and ‘setting’. We use overloaded methods to this end; the overloads cover all possible types of children. The setter part is straightforward. For instance, the illustrative model group from Sec. 3.7 (recalled below) requires the following overloads:

```
<!-- The relevant model group -->
<xs:sequence>
  <xs:element ref="tns:bar" />
  <xs:element name="bar" type="xs:int"/>
  <xs:element ref="ins:bar"/>
</xs:sequence>

// The overloaded setter method
public void Set(tns.bar x) { _1=x; } // <tns:bar>
public void Set(bar x) { _2=x; } // local <bar>
public void Set(ins.bar x) { _3=x; } // <ins:bar>

// The underlying state
private tns.bar _1;
private bar _2; // bar is a local type
private ins.bar _3;
```

(Repeating and optional particles require a special treatment, which we do not discuss here.) The getter part requires more work. We may attempt an overloaded getter method such that the overloads vary in the *result* type. However, mainstream languages like Java and C# do not support such overloading. We may also attempt to set up the getter method with a type parameter (in the sense of generics). However, (bounded) parametric polymorphism would make the getter method too polymorphic; essentially, type checking would be unable to separate legal types of children from arbitrary schema-derived object types. Hence, we need to parameterize the getter method such that an *argument type* can drive access. However, we can not (and do not want to) pass a value of the type

²² A note on .NET: a namespace is essentially a scope for declarations of classes, interfaces, and others. It is very similar to package or module scopes in other languages.

for which the getter is supposed to return a value. Instead, we pass a *proxy* for the type. Here is a generic class of proxies:

```
public class Proxy<X>
{
    protected Proxy() { }
    public static Proxy<X> proxy { get { return null; } }
}
```

The type parameter X of `Proxy` is a ‘phantom’, i.e., it is not used by the class; it is only needed for fabricating distinctive types — one for each type of children. We do not even need to populate any type `Proxy<X>`; the proxy types do not serve any run-time purpose. Instead, they are only needed for overloading resolution at compile time. Let us now assume that each schema-derived object type X implements a static member `proxy` of type `Proxy<X>` so that proxies can be conveniently picked up from the types of interest. In the running example, we can invoke the overloaded getter as follows (assuming an object `myFoo` of the schema-derived object type):

```
// myFoo is of type tns.foo
Console.WriteLine(myFoo.Get(tns.bar.proxy)); // <tns:bar>
Console.WriteLine(myFoo.Get(tns.foo.bar.proxy)); // local <bar>
Console.WriteLine(myFoo.Get(ins.bar.proxy)); // <ins:bar>
```

A.2 Compile-time validation for construction

Sec. 3.9 clarified that the common model of object construction and initialization is limited with regard to the statically checked validity of the constructed XML trees. We will now engage in a sophisticated encoding scheme that allows us to recover some static typing. The scheme is inspired by work on XML processing in Haskell using its type-class system [56,57]. Fig. 44 shows a small (contrived) schema that explores regular expression types and mixed content. The figure also shows a sample instance whose construction we hope to validate.

Let us look at *run-time validation* for inspiration. Each content model can be mapped to a finite-state machine (FSM) that performs ‘shallow validation’, i.e., the FSM checks element names (by means of instance-of tests) without descending into subtrees. Fig. 45 implements shallow validation for the stress test from Fig. 44; we assume a node-label-preserving mapping. Shallow validation is performed by the ‘untyped’ constructor of a schema-derived object type; cf. the `params` keyword.²³ Shallow validation commences as follows. There is an enum type with the states of the FSM. The nodes are passed as input to the FSM. Text nodes are accepted regardless of state because of the mixed content model. Validation throws when FSM simulation gets stuck or the input ends in a non-final state.

We can *move FSM simulation from run-time to compile-time* as follows:

²³ A note on C#: the `params` keyword enables open-ended argument lists of the array’s item type. For instance, `params object[] nodes` means that any number of arguments of type `object` is admitted and collected in a single array `nodes`.



Fig. 44. A stress test for XML-tree construction

- The states of the FSM become designated, distinct types.
- Each state type is a wrapper type around the constructed type.
- There is a static method, `New`, to initiate validating construction.
- There is an overloaded `Add` method modeling the state transitions.
- There are implicit casts from final-state types to the constructed type.

The key idea is that the overloaded `Add` method enables the type-checked construction of objects by method chaining. Fig. 46 illustrates shallow, compile-time validation for the running example. The shown method chain resembles the original XML literal.

A pure FSM approach suffers from obvious scalability problems. Occurrence constraints (such as `maxOccurs="42"`) cannot be checked efficiently with a plain FSM because of the number of states that would be needed for counting. However, in practice non-trivial occurrence constraints (other than `'?'`, `'+'`, `'*'`) hardly occur [32]. More seriously, whenever content models prescribe free order, then a pure FSM approach would explode due to the number of permutations to be considered. Free order applies to attribute

```

public class foo
{
    // Content
    private object[] nodes;

    // States of FSM
    public enum State { BeforeA, BeforeB, SeenB, BeforeChoice };

    // Run-time validating constructor
    public foo(params object[] nodes)
    {
        this.nodes = nodes;
        State s = State.BeforeA;
        foreach (var o in nodes)
            if (!(o is string))
                switch (s) {
                    case State.BeforeA :
                        if (o is a) { s++; break; }
                        goto default;
                    case State.BeforeB :
                        if (o is b) { s++; break; }
                        goto default;
                    case State.SeenB :
                        if (o is b) break;
                        if (o is a) { s++; break; }
                        if (o is c) { s++; break; }
                        goto default;
                    case State.BeforeChoice :
                        goto default;
                    default:
                        throw ...;
                }
            if ((s!=State.SeenB) && (s!=State.BeforeChoice))
                throw ...;
    }
}

// classes a, b, c omitted

```

Construction sample

```

var myFoo = new foo(
    "Text before 'a'. ",
    new a(),
    "Text before 1st (mandatory) 'b'. ",
    new b(),
    "We could stop anywhere from here on.",
    new b(),
    new b(),
    new c(),
    "Let's stop here, indeed.");

```

Fig. 45. Shallow, run-time validation for the content model in Fig. 44

sets and <all> composites. We would need more advanced techniques and type-system support [3] to recover from this problem.

```

public class foo : List<object>
{
    // Default construction in begin state
    public static foo.BeforeA New { get { return new BeforeA(new foo()); } }

    // States with transitions
    public struct BeforeA
    {
        internal BeforeA(foo v) { it = v; }
        public BeforeA Add(string v) { it.Add(v); return this; }
        public BeforeB Add(a v) { it.Add(v); return new BeforeB(it); }
        internal foo it;
    }
    public struct BeforeB
    {
        internal BeforeB(foo v) { it = v; }
        public BeforeB Add(string v) { it.Add(v); return this; }
        public SeenB Add(b v) { it.Add(v); return new SeenB(it); }
        internal foo it;
    }
    public struct SeenB
    {
        internal SeenB(foo v) { it = v; }
        public SeenB Add(string v) { it.Add(v); return this; }
        public SeenB Add(b v) { it.Add(v); return this; }
        public BeforeChoice Add(a v) { it.Add(v); return new BeforeChoice(it); }
        public BeforeChoice Add(c v) { it.Add(v); return new BeforeChoice(it); }
        public foo End { get { return it; } }
        internal foo it;
    }
    public struct BeforeChoice
    {
        internal BeforeChoice(foo v) { it = v; }
        public BeforeChoice Add(string v) { it.Add(v); return this; }
        public foo End { get { return it; } }
        internal foo it;
    }

    // End states deliver the completed object
    public static implicit operator foo(SeenB v) { return v.it; }
    public static implicit operator foo(BeforeChoice v) { return v.it; }
}

```

Construction sample

```

foo myFoo = foo.New
    . Add("Text before 'a'.")
    . Add(new a())
    . Add("Text before 1st (mandatory) 'b'.")
    . Add(new b())
    . Add("We could stop anywhere from here on.")
    . Add(new b())
    . Add(new b())
    . Add(new c())
    . Add("Let's stop here, indeed.");

```

Fig. 46. Shallow, compile-time validation for the content model in Fig. 44

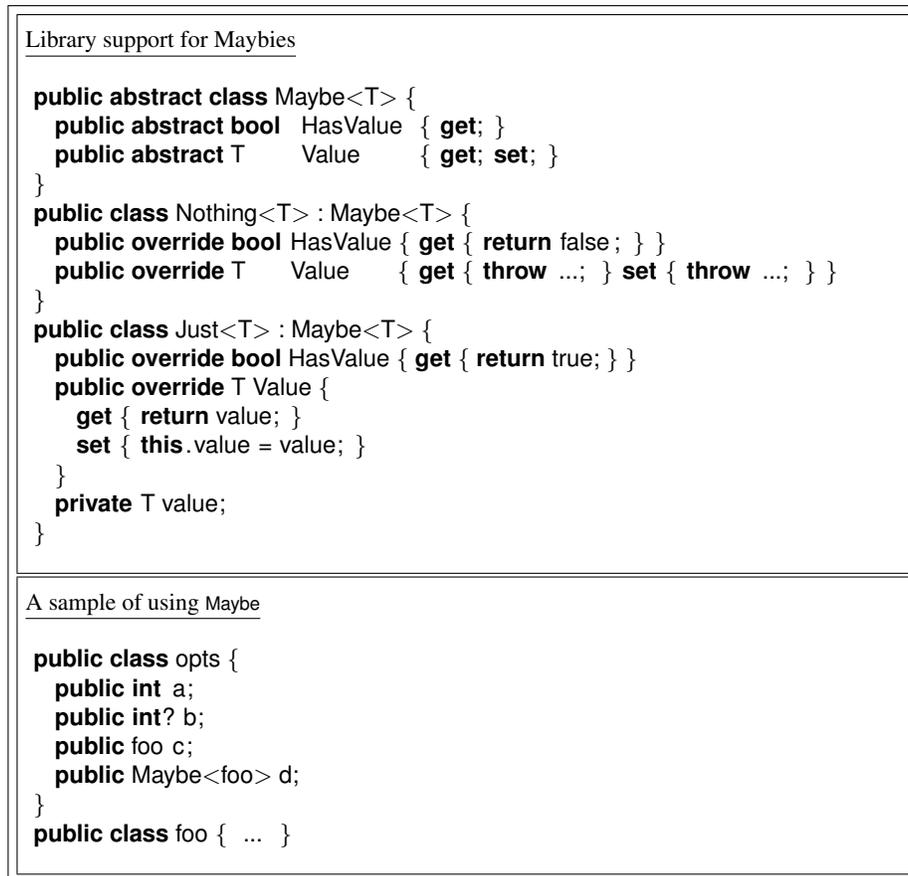


Fig. 47. A blend of Haskell's Maybe and .NET's Nullable

A.3 Haskell-like maybies

Sec. 4.1 clarified that optionality of reference types is not discoverable by default. We could explicitly express optionality by the use of a designated type constructor, very much like .NET's existing type constructor `Nullable`, but without its restriction to value types. Fig. 47 defines such a type constructor; we use the name `Maybe` in reference to Haskell's standard type constructor for optionality.

The figure also illustrates the use of `Maybe` for the running example of Sec. 4.1. In principle, we could use `Maybe` in place of "?", even for value types. However, .NET's nullable types are linguistically richer and more efficient. In particular, primitive operations on the value types are lifted to nullable types, and some forms of boxing are optimized away.

One minor problem with the simple optionality technique is that the concept of 'null' has become ambiguous: the code may need to be prepared to find both the normal

null and `Maybe`'s `Nothing`. This problem can be mitigated by switching from a field-based model to a property-based model with getters that never return normal nulls. Thus:

```
public Maybe<foo> d {
    get { if (_d==null) _d = new Nothing<foo>(); return _d; }
    set { _d = value; }
}
private Maybe<foo> _d;
```

Obviously, we should use the singleton design pattern to avoid creating many instances of `Nothing<T>` for any given type. A more challenging source of overhead concerns the use of wrapping with `Just`; one extra object per XML subtree in an optional position would be created. These costs may be substantial. This problem can be mitigated by the use of interface polymorphism, as shown below.

In Fig. 48, we use an interface, `IMaybe`, instead of the earlier class `Maybe`. Wrapping with `Just` is unnecessary because each and every schema-derived class implements the `Just`-like behavior — either locally or, when possible, by subclassing the class `Just`. We also show again property implementations; one for a mandatory particle; another one for an optional particle. The property implementation for the mandatory particle refuses the normal null. The getter implementation for the optional particle translates the normal null to `Nothing`.

A.4 Generics for compositors

Sec. 4.3 suggested that XSD compositors may be modeled as generic classes, thereby providing one mapping option for nested composites. Fig. 49 defines a generic class for binary choices, in fact, binary, type-indexed co-products [53]. The class covers the following idioms: (i) construction such that one must commit to either branch of the choice; (ii) updates on the grounds of ‘actions’ to be performed on the object of the choice; (iii) queries on the grounds of ‘functions’ to be applied to the object of the choice. Further operations may be added, e.g., for cloning. Conceptually, choices are commutative, but the generic class is somewhat restrictive in this respect. That is, the textual order of the type parameters in `Choice<X1,X2>` is also assumed for the argument lists of `Do` and `Apply`. (We could attempt to enable all possible argument orders, except that this idea does not scale for choices with more branches.)

For `<all>` composites, we could use the dual concept of type-indexed co-products, i.e., type-indexed products. We omit this variation. Fig. 50 defines a (trivial) generic class for binary sequences. The class essentially corresponds to the type constructor for pairs. There are getters and setters for both components of the sequence. Construction must enumerate the components of the sequence. Order of the particles is relevant for sequences, and hence, we provide a positional access protocol based on members `First`, `Second`, ... (as opposed to a type-driven protocol).

<p>An interface for optionality</p> <pre> public interface IMaybe<T> { bool HasValue { get; } T Value { get; } } </pre>
<p>A generic Nothing</p> <pre> public class Nothing<T> : IMaybe<T> { public bool HasValue { get { return false; } } public T Value { get { throw ...; } set { throw ...; } } } </pre>
<p>A generic Just</p> <pre> public class Just<T> : IMaybe<T> where T : Just<T> { public bool HasValue { get { return true; } } public T Value { get { return (T)this; } } } </pre>
<p>A sample of using Maybe</p> <pre> public class opts : Just<opts> { // Reusable singleton internal static IMaybe<opts> Nothing = new Nothing<opts>(); public int a; public int? b; public foo c { get { if (_c==null) throw ...; return _c; } set { if (value==null) throw ...; _c = value; } } public IMaybe<foo> d { get { if (_d==null) _d = global::d.Nothing; return _d; } set { _d=value; } } private foo _c; private IMaybe<foo> _d; } </pre>

Fig. 48. Interface-based optionality

```

public class Choice<X1,X2>
{
    // Injection by functional construction
    public Choice(X1 v) { any=v; idx=1; }
    public Choice(X2 v) { any=v; idx=2; }

    // Perform action
    public void Do(Action<X1> a1, Action<X2> a2) {
        switch (idx) {
            case 1: a1((X1)any); break;
            case 2: a2((X2)any); break;
        }
    }

    // Apply function
    public R Apply<R>(Func<X1,R> f1, Func<X2,R> f2) {
        switch (idx) {
            case 1: return f1 ((X1)any);
            case 2: return f2 ((X2)any);
            default: throw ...; // impossible exception
        }
    }

    // Private state
    private object any;
    private int idx;
}

```

Relevant delegate types

```

public delegate void Action<T> (T it) // Actions that effect argument
public delegate Y Func<X,Y>(X x); // Single –argument functions

```

Fig. 49. A generic class for binary choices

```

public class Sequence<X1,X2>
{
    // Functional construction
    public Sequence(X1 x1, X2 x2) { _x1=x1; _x2=x2; }

    // Getters and setters
    public X1 First { get { return _x1; } set { _x1 = value; } }
    public X2 Second { get { return _x2; } set { _x2 = value; } }

    // Private state
    private X1 _x1;
    private X2 _x2;
}

```

Fig. 50. A generic class for binary sequences