Mappings make data processing go ’round
An inter-paradigmatic mapping tutorial

Ralf Lämmel Erik Meijer

Microsoft Corp., Data Programmability Team, Redmond, USA

Abstract. Whatever programming paradigm for data processing we choose, data has the tendency to live on the other side or to eventually end up there. The major paradigms for data processing are Cobol, object, relational and XML; each paradigm offers many facets and many versions; each paradigm provides specific forms of data models (object models, relational schemas, XML schemas, etc.). Each data-processing application depends on a horde of interrelated data models and artifacts that are derived from data models (such as data-access layers). Such conglomerations of data models are challenging due to paradigmatic impedance mismatches, performance requirements, loose-coupling requirements, and others. This ubiquitous problem calls for a good understanding of techniques for mappings between data models, actual data, and operations on data. This tutorial lists and discusses mapping scenarios, mapping techniques, impedance mismatches and research challenges regarding mappings.

Keywords: Data processing, Mapping, XML data binding, Object-XML mapping, Object-relational mapping, Cross-paradigm impedance mismatch, Data modeling, Data access, Loose coupling, Software evolution

Table of Contents

1 Introduction . . . 3
2 Mapping examples . . . 4
  2.1 From concrete to abstract syntax . . . 4
  2.2 Data binding in user interfaces . . . 8
  2.3 XML data binding . . . 12
3 Mapping concepts . . . 14
  3.1 Universal representations . . . 15
  3.2 Canonical mappings . . . 16
  3.3 Mapping customization . . . 16
  3.4 Type- vs. instance-based . . . 17
  3.5 The programmatic-to-declarative scale . . . 19
  3.6 Annotations vs. references . . . 21
  3.7 Updateability . . . 23
  3.8 Usage protocols . . . 26
  3.9 Further reading . . . 27
4 Cross-paradigm impedance mismatches . . . 27
  4.1 Characteristics of major paradigms . . . 27
  4.2 An open-ended list of mapping issues . . . 31
  4.3 Exemplar frictions . . . 33
5 Call to arms . . . 36
  5.1 Overall goals . . . 36
  5.2 A list of challenges . . . 36
  5.3 Challenges in detail . . . 37
6 Concluding remarks . . . 41
A Exercises and riddles . . . 47
  A.1 Mappings in parsing and un-parsing . . . 48
  A.2 Mappings for XML grammars . . . 49
  A.3 Compensation of semantical impedance mismatches . . . 50
  A.4 XML, object, relational mapping . . . 51

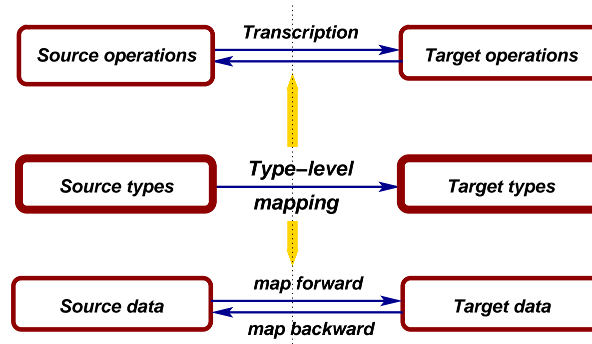


Fig. 1. Levels in mapping

1 Introduction

We steal the beginning of our tutorial from elsewhere: *"Once upon a time it was possible for every new programmer to quickly learn how to write readable programs to Create, Read, Update and Delete business information. These so-called CRUD applications, along with reporting, were pervasive throughout business and essentially defined IT or MIS as it was called in those days."* [92] (Dave Thomas: "The Impedance Imperative Tuples + Objects + Infosets = Too Much Stuff!").

Instead, today we face the following diversity:

- Cobol applications with keyed files are still developed and they make sense.
- Relational databases have fully matured and they are unarguably omnipresent.
- OO databases innovate, perhaps at a slow pace, but they must be taken seriously.
- The XML hype is over. XML types and XML documents are everywhere now.
- All these paradigms have triggered a myriad of query languages and 4GL tools.
- Much current CRUD development is done with OO languages with various APIs.

This tutorial is about the challenges implied by such diversity in data modeling and data processing. Either there are respectable, perhaps fundamental reasons for all this diversity, or it is just plain IT reality. No matter what, we need to map amongst these paradigms, and everyone is trying to do that anyhow. According to a designated online resource[1], there are roughly 60 established products for X/O mapping, also known as XML data binding, i.e., XML schemas or DTDs are mapped to object models. We reckon that practice is ahead of foundations in this area, but this surely implies ad-hoc approaches with unnecessary limits and complexities. We need basic and applied research on inter- and intra-paradigm mappings.

What is a mapping anyway?

We should make more precise what we mean by 'mapping'. We have to disappoint those readers looking for a detailed or even formal definition. Instead we offer the following explanation and the illustration in Fig. 1.

[1] http://www.rpbourret.com/xml/XMLDataBinding.htm

– Mapping is essentially about the transformation of values between data models.[2]
– The data models typically involve different paradigms (Cobol, OO, relational, XML).
– Fig. 1 opts for a type-based mapping (described at the type level).
– By contrast, instance-based mappings directly define value transformations.
– Other mappings may implicitly define data models for source and target.
– CRUD operations may need transcription from the source to the target or vice versa.
– There may be more levels than those in the figure, e.g., the level of protocols.

Road-map for the tutorial

– Sec. 2 presents diverse illustrative *mapping examples*.
– Sec. 3 is an attempt to collect (some) *mapping concepts*.
– Sec. 4 reveals *impedance mismatches* for inter-paradigmatic mappings.
– Sec. 5 calls to arms regarding *engineering and research challenges*.
– Sec. 6 concludes the article.
– The appendix collects a good number of exercises.

2 Mapping examples

We will walk through a few data-processing scenarios that involve mappings. We strive for diversity so that we show the ubiquitousness of the mapping notion in programming and software development. As we go, we hint at established techniques, typical requirements and recurring problems.

2.1 From concrete to abstract syntax

Language processing, including compiler construction, involves mappings in abundance. Most notably, a parser needs to map concrete syntax to reasonable parse trees or proper ASTs (i.e., abstract syntax trees). In fact, a non-trivial, well-organized language processor may involve several abstract syntaxes related to different components in front- and middle-ends. Yet other mappings in language processors can be concerned with immediate representations such as PDG and SSA [28,20]. We will discuss some forms of mapping concrete syntax to parse trees or ASTs.

Concrete syntax as mapping source Consider the following ANTLR[3] grammar for the concrete syntax of arithmetic expressions.[4] The actual encoding represents operator priorities by 'grammatical layers' — as it is common for top-down parsing. That is, expression forms are grouped per operator priority using an auxiliary nonterminal for all groups — except the top-most one:[5]

[2] The term 'data model' is ambiguous as it may refer to both the general data model of a paradigm such as the 'relational model'; it may also refer to domain-/application-specific data models such as a particular 'relational schema' or 'object model'; http://en.wikipedia.org/wiki/Data_model. In this tutorial, we favor the latter meaning.

[3] ANTLR web site: http://www.antlr.org/

[4] Source: http://www.bearcave.com/software/antlr/antlr_examples.html

[5] We will start code fragments with a comment that identifies the used programming language.

```
// ANTLR grammar
expr          :  mul_expr (addOP mul_expr)* ;
mul_expr      :  sign_expr  (mulOP sign_expr)* ;
sign_expr     :  (MINUS)? primary_expr ;
primary_expr  :  IDENT
              |  constant
              |  ( LPAREN! expr RPAREN! ) ;
```

Option: untyped, canonical mapping ANTLR offers the option to construct parse trees in a canonical manner using a language-independent format (which is a sort of universal representation type). The problem with such a generic approach is that no abstraction is carried out (in the sense of ASTs), and no typing discipline for parse trees is enforced. So we are seeking different mapping options.

Option: mapping in 'all detail' The attribute grammar paradigm [55,81] can be used for a mode of parse-tree construction that improves on the above-mentioned problems. All parser generator tools like ANTLR (and Yacc, PRECC, BTYACC, etc.) support this technique (with more or less strong typing). ANTLR presupposes type declarations for the intended parse-tree format. The actual mapping has to be described in the parser specification: semantic actions synthesize parse-tree fragments.

The following ANTLR snippet is a refinement of the last context-free production in the earlier grammar for expressions. The added semantic actions build a binary expression from two operands and an operator.

```
// ANTLR production with in-lined C++ code
expr returns [binaryNode a_expr]
  { exprNode m1 = NULL;
    exprNode m2 = NULL;
    opNode op = NULL;
  }
  : m1 = mul_expr { a_expr = m1; }
    ( op = addOP m2 = mul_expr
    { a_expr = new binaryNode( op, a_expr, m2 ); } )*
  ;
```

We omit the declarations for the referenced C++ classes: exprNode (the abstract base class for expressions), binaryNode, opNode. A problem with this approach is that it is 'a lot of work'. First, the abstract syntax has to be worked out in all tedious details, even though it may be 'intentionally' similar to the concrete syntax. Second, the mapping has to be 'coded' in all detail, again without leveraging any similarities between concrete and abstract syntax. Furthermore, we end up with a poor separation of concerns in so far that the original context-free productions get invaded by declarations and semantic actions for parse-tree construction.

Option: generative mapping One can improve on these problems by means of generative programming [23], namely a grammar-oriented form of it; cf. [50,57,9,40] for

related work. (We need meta-grammarware according to [54].) We briefly summarize an approach actually offered by an existing technology: GDK (the Grammar Deployment Kit [57]). That is, GDK processes pure grammars (without any semantic actions) and generates a typed parse-tree format as well as the bloated parser specification that comprises the tedious mapping. The type declarations for parse-tree formats are also valuable for consumers of the constructed parse trees. The generated parser specification can be processed by a conventional parser generator. Various programming languages and parser generators are supported. Here is an example of a generated Yacc [49] production for parenthesized expressions.

```
// Generated Yacc production with embedded C code
expr_in_parens
 : T_QPOPEN
   expr
   T_QPCLOSE
   { $$ = build_expr_in_parens($1, $2, $3); }
 ;
```

The function symbol build_expr_in_parens is one of the term constructors that is generated from the pure grammar. Consumers of the parse trees can use accordingly generated matching functions. This approach still does not solve the problem of abstraction in the sense that the constructed parse trees mirror (too) precisely the concrete syntax.

Option: simplify concrete into abstract syntax There exist declarative mapping approaches such that the abstract syntax and the mapping from concrete to abstract syntax can be controlled more explicitly without switching to the other extreme of defining the mapping in all detail — as it was the case for the attribute-grammar approach, unfortunately. We will discuss a particularly advanced approach that is supported by the compiler generator Eli [35]. That is, Eli provides designated tool support, Maptool [51], for concrete-to-abstract syntax mappings.

Consider the following context-free syntax using Eli's grammar notation:[6]

```
// Eli's grammar notation
Program      ::=  Statement + .
Statement    ::=  Computation ';' .
Computation  ::=  Expr/LetExpr/WhereExpr .
LetExpr      ::=  'let' Definitions 'in' Expr .
WhereExpr    ::=  Expr 'where' Definitions .
Definitions  ::=  Definition // ',' .
Definition   ::=  Identifier '=' Expr .
Expr         ::=  Expr '+' Term / Expr '−' Term / Term .
Term         ::=  Term '∗' Factor / Term '/' Factor / Factor .
Factor       ::=  '−' Factor / Primary .
Primary      ::=  Integer / Identifier / '(' Computation ')' .
```

[6] Eli uses an EBNF-like notation. That is, '+' is for repetition (i.e., lists), and '/' is for alternatives (elsewhere denoted as '|'). There is notation for separator lists: '$e//c$', where e is the phrase to be repeated and c is the character for separation.

As in the earlier example, there are several layers of expressions: Computation, Expr, Term, Factor, Primary. These layers are biased towards parsing concrete syntax while only adding irrelevant complexity to subsequent phases. Hence, we would prefer to unite these layers in the abstract syntax. This is accomplished by the following fragment of a Maptool mapping specification:

```
// Eli's Maptool notation
MAPSYM
 Expr ::= Computation Term Factor Primary .
```

That is, the various nonterminals on the right-hand side of the MAPSYM declaration are placed in an *equivalence class*, which effectively implies an abstract syntax as if Expr were defined by a flat list of alternatives. This simplification enables more concise language processing code. For instance, expression evaluation does not need to handle all the concrete syntactical variations implied by the different nonterminals.

Option: refine abstract into concrete syntax Rather than simplifying the concrete syntax such that a suitable abstract syntax is derived, we can also start from a simple abstract syntax and refine it into the existing concrete syntax — thereby defining a mapping. Let us design an abstract syntax that is as abstract and suggestive as it could be for the purpose of, say, *name analysis*. (In compiler construction, name analysis tends to refer to the concept of resolving (or better: establishing) the links between using (referring) occurrences and defining (declaring) occurrences.)

```
// Eli's (abstract) grammar notation
Program              LISTOF Statement
BoundExpr     ::=    Definitions Expr
Definitions          LISTOF Definition
Definition    ::=    IdDef '=' Expr
Primary       ::=    IdUse
IdDef         ::=    Identifier
IdUse         ::=    Identifier
```

It happens that several of the nonterminals in the abstract syntax correspond to nonterminals in the concrete syntax. Hence, they are automatically mapped by 'name coincidence'. However, there are major idioms for completing name coincidence into a concrete-to-abstract syntax mapping, which we will discuss in the sequel.

The abstract sort BoundExpr does not have an immediate counterpart in the concrete syntax. We use it as a general form of a binding group (say definitions). In fact, BoundExpr is meant as an abstraction for let and where expressions. This intent can be expressed by the following bits of mapping specification:

```
// Eli's Maptool notation
MAPSYM
 BoundExpr ::= LetExpr WhereExpr .
```

```
MAPRULE
  LetExpr      ::=  'let' Definitions 'in' Expr <$1$2> .
  WhereExpr    ::=  Expr 'where' Definitions <$2$1> .
```

That is, the nonterminals LetExpr and WhereExpr are placed in an equivalence class with BoundExpr. The productions for LetExpr and WhereExpr are associated with directions for AST construction. (The phrases `<$1$2>` and `<$2$1>` express the subtrees of the AST in terms of indexes of the subtrees of the concrete parse tree.)

The abstract sorts IdDef and IdUse *partition* the nonterminal Identifier from the concrete syntax. The distinct nonterminals are used for defining vs. using occurrences of Identifier . The nonterminal Definition is defined in both grammars, while the abstract syntax points out that the occurring identifier is actually a defining occurrence. This mapping leads to a useful abstract syntax design because it enables a language-independent name analysis. That is, the name analysis can identify the defining vs. using role of an identifier solely by means of the nonterminal symbols IdDef and IdUse. Without this distinction, the name analysis would need to have intimate knowledge about grammar productions and positions in which identifiers occur in this or that role.

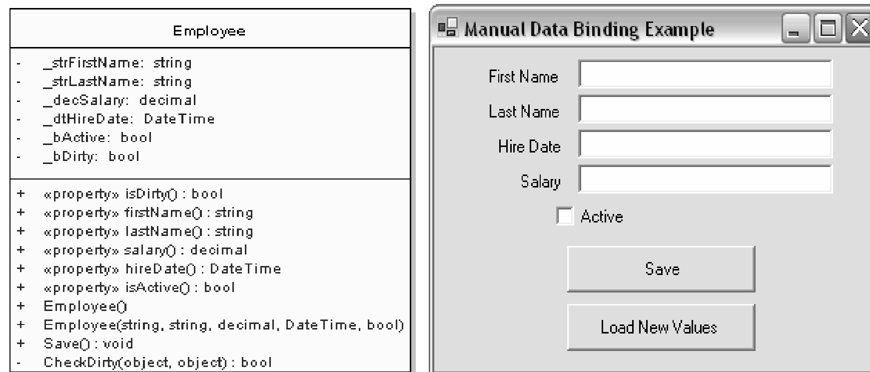
2.2 Data binding in user interfaces

Interactive applications require mappings of the kind that application data is bound to user-interface elements.[7] In the small, an archetypal example would be about associating a field, such as the first name of an employee object, to the text property of a text box in a form. The term (GUI) 'data binding' is nowadays used for this problem, but the overall issue is not tied to modern platforms such as Java and .NET. For instance, forms-based Cobol applications have dealt with the same problem for ages: application data must be mapped to user-interface elements of screens (or forms), user-input validation has to be carried out, and a protocol for change notification must be provided.

Option: point-to-point programmatic mapping Let us consider an example of GUI data binding. In Fig. 2, on the left-hand side, we see the class structure of an employee object; on the right-hand side, we see a GUI form for operating on an employee object. Let us also assume controls for the various fields and buttons:

```
// C# 1.0 code (using System.Windows.Forms)
public class myForm : Form
{
  private TextBox txtFirstName;
  private TextBox txtLastName;
  private TextBox txtHireDate;
  private TextBox txtSalary;
  private CheckBox chkIsActive;
  private Button btnLoadNewValues;
  private Button btnSave;
```

[7] A comment on terminology: 'data binding' is often implicitly taken to mean 'binding data to a GUI'. For an unambiguous terminology, we say 'GUI data binding'.



Source: http://www.15seconds.com/issue/040908.htm

Fig. 2. Data to be bound in a GUI

```
  // to be cont'd
}
```

Our application data is stored in a field like this:

```
private Employee _oEmployee = null;
```

A simple approach ('brute force') to data binding commences as follows, the binding (or mapping) code boils down to two explicit *move* routines; one to fill the form with application data (i.e., the content of the employee field); another to save the content of the form. In both directions, we define a kind of *point-to-point* mapping:

```
// Exception-handling code omitted
private void DataToForm()
{
  this.txtFirstName.Text     = _oEmployee.firstName;
  this.txtLastName.Text      = _oEmployee.lastName;
  this.txtSalary.Text        = _oEmployee.salary.ToString();
  this.txtHireDate.Text      = _oEmployee.hireDate.ToShortDateString();
  this.chkIsActive.Checked   = _oEmployee.isActive;
}

private void FormToData()
{
  _oEmployee.firstName  =  txtFirstName.Text;
  _oEmployee.lastName   =  txtLastName.Text;
  _oEmployee.salary     =  Convert.ToDecimal(txtSalary.Text);
  _oEmployee.hireDate   =  Convert.ToDateTime(txtHireDate.Text);
  _oEmployee.isActive   =  chkIsActive.Checked;
}
```

This coding style is well in line with common practice. There is one striking weakness of this approach: we end up coding the mapping *twice*. Another weakness is that we code conversions allover the place and thereby bypass static type checking; cf. the use of Convert.ToDecimal.

Option: point-to-point mapping declarations We can improve on this brute-force approach by exploiting the designated data-binding interface of GUI controls. That is, we can actually inform each and every control about the associated application data:

```
private void MyBind()
{
  txtFirstName.DataBindings.Add("Text", _oEmployee, "firstName");
  txtLastName.DataBindings.Add("Text", _oEmployee, "lastName");
  txtSalary.DataBindings.Add("Text", _oEmployee, "salary");
  Binding bindHireDateText = new Binding("Text", _oEmployee, "hireDate");
  bindHireDateText.Format +=
     new ConvertEventHandler(DateTimeToShortDateString);
  txtHireDate.DataBindings.Add(bindHireDateText);
  chkIsActive.DataBindings.Add("Checked", _oEmployee, "isActive");
}
```

As a result, the mapping is specified only *once*, and the amount of conversion code is restricted to cases in which defaults are not sensible; cf. DateTimeToShortDateString. Unfortunately, we have to pay a considerable price for the improvement of conciseness. The mapping description is largely string-based:

– "Text" vs. this.txtFirstName.Text,
– "firstName" vs. _oEmployee.firstName.

Hence, one dimension of subsequent improvement is to provide static typing for such mappings, but we will first consider a more operational issue. The trouble is that the form is filled only *initially* when the binding is issued, but subsequent changes of the application data are not passed on to the form. Updating only works one way: changes in the form are mapped back to the bound setters, but not vice versa.

Two-way updates More generally, a mapping approach might need to maintain some degree of bi-directional tracking between source and target. In this example of GUI data binding, change propagation can be arranged as follows. The data bindings of Windows.Forms can be made to listen to changes in the application data. The relevant idiom is that a setter on application data should trigger a change event:

```
public class Employee
{
  // The private field for application data
  private string _firstName;

  // An event for changes on _firstName
```

Initial state **State after user interaction**

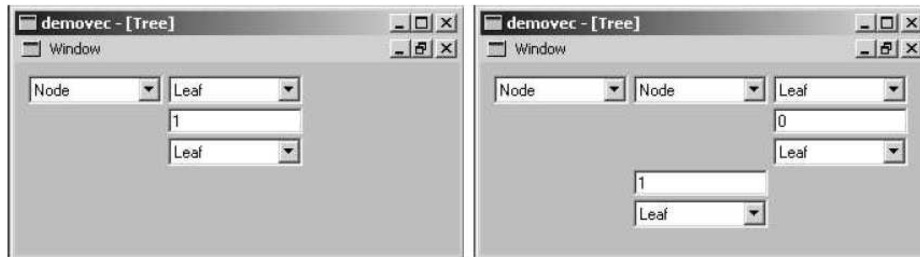


Fig. 3. GUI states with different trees (adopted from [2])

```
  public event EventHandler firstNameChanged;

  // The firstName property; note the setter
  public string firstName
  {
    get { return _firstName; }
    set {
      _firstName = value;
      firstNameChanged(this, new EventArgs());
  }}

  // ... likewise for other data ...
}
```

Change tracking is name-based: the name of a bound property + ‘Changed’ is the name of the event (if defined by the programmer) observed by the GUI data binding framework so as to learn about state changes; cf. the couple firstName and firstNameChanged in the code snippet. Various GUI frameworks leverage similar idioms. It is also common to leverage design patterns that help modeling some aspects of update protocols and consistency checking; e.g., the observer design pattern [33], the model-view-controller architecture [59], and friends.

Typed *and* canonical *and* customizable *and* live mapping Regarding the remaining typing weakness, we would like to contrast Windows.Forms with a strongly typed approach that uses the modern functional language Clean [2,1]. At the same time, this approach also illustrates a callback-based technique for two-way change tracking. So the bound GUI is always in sync with the data layer.

The ambition of the Clean-based approach is to allow for editing data in a highly systematic manner. To this end, a generic programming approach (in the sense of induction on type structure) is employed. The generated GUI controls are called GECs — Graphical Editor Components. The overall assumption is that a reasonable GEC for a specific value v can be constructed just by observing the structure of v’s type. In Fig. 3, the GECs for two values are shown. The left GEC represents a binary, node-labeled tree of the form Node Leaf 1 Leaf. When the user changes the upper Leaf to Node, through

the pull-down menu, the GEC evolves as shown on the right-hand side of the figure. *Any* GEC is constructed via the following Clean function mkGEC.

```
-- A generic Clean function (looks like Haskell, almost)
generic mkGEC t ::  [GECAttribute]            -- Control appearance
                    t                         -- The initial value
                    (CallBackFunction t ps)   -- Call back for changes
                    (PSt ps)                  -- Program state
                 -> (GEC t (PSt ps),PSt ps)   -- Constructed GEC + state
```

The type of the function hints at the status of the mapping to be canonical and customizable and live. The canonical mapping status is implied by the fact that mkGEC is a generic (polytypic) function with the type parameter t. The customization capability is modeled by the first argument that anticipates a list of attributes that control the appearance of the GEC. The live status of a GEC is implied by the fact that its creation must define an initial value (cf. second argument) and a CallBackFunction to be invoked when the edited value is changed (no matter whether the change is caused by editing or by programmatic access). The constructed GEC also provides read and write access to the bound value. (The rest of the function's signature deals with the fact that GEC construction and GEC usage involves state transformation. In Haskell terms, we would expect some use of the IO or state monad.)

2.3 XML data binding

The term 'XML data binding' [70,15] refers to the problem of providing an object model that is meant to represent an XML schema (an XSD description) in the object world (or vice versa). This is a modern mapping scenario in which either an XML schema or an object model is given, and the counterpart (i.e., the object model or the XML schema) is to be derived. In the subsequent illustrations, we are going to use an XML schema sample for widgets (rectangle, squares, circles), as they may occur in a drawing application:[8]

```
<!-- XML schema -->
<xs:element name="Widgets">
 <xs:complexType>
  <xs:choice minOccurs="0" maxOccurs="unbounded">
   <xs:element name="Rectangle" type="Rectangle"/>
   <xs:element name="Square" type="Square"/>
   <xs:element name="Circle" type="Circle"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

[8] This example explores the XSD variation on the ingenious 'shapes example' — an OO benchmark that has been designed by Jim Weirich and deeply explored by him and Chris Rathman. See the code collections http://onestepback.org/articles/poly/ and http://www.angelfire.com/tx4/cus/shapes/.

```
<xs:complexType name="Rectangle">
 <xs:sequence>
  <xs:element name="XPos" type="xs:int"/>
  <xs:element name="YPos" type="xs:int"/>
  <xs:element name="Width" type="xs:int"/>
  <xs:element name="Height" type="xs:int"/>
 </xs:sequence>
</xs:complexType>

<!-- ... Square elided ... -->
<!-- ... Circle elided ... -->
```

While this a perfectly reasonable XML schema, we may encounter challenges when mapping this schema to objects. There exist many different XML-data binding technologies; each technology defines its own canonical mapping (and one may argue at times which one is better).

Schema-derived classes The following class has been generated by the .NET 2.0 technology xsd.exe. The fields of class Rectangle resemble the structure of the corresponding complex type definition. The XSD simple type xs:int is mapped to the VB.NET type Integer.

```
' VB.NET 8.0 code
' Note: all generated custom attributes omitted
Partial Public Class Rectangle
 Private xPosField As Integer
 Private yPosField As Integer
 Private widthField As Integer
 Private heightField As Integer
 Public Property XPos() As Integer
  Get
   Return Me.xPosField
  End Get
  Set
   Me.xPosField = value
  End Set
 End Property
 ' ... other properties elided ...
End Class
```

Adaptation of mapping results xsd.exe's XML-to-object mapping is fully canonical; there are no means of influencing the mapping. However, one may adapt the mapping *result*, as we will discuss. Suppose we want to process collections of shapes by exploiting subtype polymorphism such that the executed functionality (e.g., for drawing) is specific to the kind of shape. So we want the classes Rectangle, Square and Circle to engage in a subtype hierarchy rooted by a new class, say Shape:

```
' A base class for all shapes
Public MustInherit Class Shape ' Abstract class
  Public MustOverride Sub draw() ' Abstract method
End Class
```

How can we make it so that Rectangle etc. inherit from Shape and implement draw? A naive and problematic approach would be to manually adapt the generated classes. Adapting generated code is almost universally a bad idea for obvious reasons. It turns out that we can employ linguistic means to adapt the mapping result. That is, we can use VB.NET 8.0's *partial classes*, which admit compile-time extension of classes. In particular, we can resolve the aforementioned problem without touching the generated code at all. We provide another slice of the (partial) class Rectangle; the idea is that both 'slices' are merged by the compiler.

```
Partial Public Class Rectangle
   Inherits Shape
   Public Overrides Sub draw()
      WriteLine("Drawing a rectangle.")
   End Sub
End Class
```

Dead ends in mapping Here is the generated code for Widgets:

```
' The class that corresponds to the Widgets element declaration
Partial Public Class Widgets
 Private itemsField() As Object
 Public Property Items() As Object()
   ' ... trivial implementation elided ...
 End Property
 ' ... rest of class elided ...
End Class
```

The use of normal arrays for collecting widgets is reasonable as long as we *observe* de-serialized XML content. However, should we want to add widgets, we may prefer a more 'dynamic' collection type such as List. Also, the widgets are exposed in a rather untyped manner (cf. Object). We may want to use a strongly typed, generic collection whose item type is Shape. The partial-class mechanism and other available programming idioms do not help in these cases.

One may argue that the canonical mapping at hand is simply suboptimal and needs to be improved, no matter what. However, any mapping technology must eventually adopt some mapping rules and options. There is always a chance that someone ends up wanting a different rule or another option later. Ideally, there would be a fundamental way of defending the quality and the completeness of a mapping.

3 Mapping concepts

Let us raise the level of abstraction and focus on concepts. Throughout the section, we continue discussing examples so that we can illustrate the identified concepts and

collect more data points. Ideally, we would like to deliver the perfect, comprehensive, formal and meaningful framework for the categorization and assessment of existing and new mapping approaches. We are unable to complete such a task at this point in time. Incidentally, the purpose of this tutorial is to motivate research that may enable the completion of the envisaged framework.

3.1 Universal representations

Many mapping scenarios regularly call for (or take advantage of) universal representations. This concept is based on a relationship between types in a given type language (CLR classes, Haskell data types, etc.) and a universal (fixed) representation type for that type language ('the universe'). Here are applications of universal representations:

- Serialization of data (to text or XML) for persistence.
- Serialization for interoperability using XML again.
- Type erasure for foreign-language interfacing.
- Type erasure to escape to a dynamically typed or untyped coding style.

A good example for the last item is the need to escape from strong typing in cases where a given mapping problem can be more easily addressed using the simple structure of the universal representation type. We will illustrate this scenario in a Haskell context, but similar examples could be provided in an OO context (using reflective programming). We face the following mapping pipeline:

```
-- Haskell 98 code
myMapping = tree2data        -- Step 3: get back into typed world
          . trickyMapping    -- Step 2: untyped but powerful mapping
          . data2tree        -- Step 1: get out of typed world
```

In this pipeline, typed data (i.e., Haskell terms) is first exposed in an untyped tree format (cf. data2tree); then a 'tricky' mapping can be defined without running into the limitations imposed by the type system; finally the universal representation is mapped back into strongly typed data. The last step may fail of course.

A good example of a tricky mapping is a data conversion due to *type evolution* (i.e., evolution of the data model). In this case, there are two versions of the same system of data types which only differ in some details. Strongly typed programming fails to provide a concise way of mapping version A to version B; the *verbose* way would be to exhaustively cover all types and their cases in equations of functions that define a mapping. Untyped programming makes it easy to generically process the input data and to focus on the differences between the two versions.

For clarity, these are the types of the functions involved in myMapping:

```
data2tree     :: Data a => a -> Tree String
trickyMapping :: Tree String -> Tree String
tree2data     :: Data a => Tree String -> Maybe a
```

We use n-ary, labeled trees as the universal representation type. Haskell's standard libraries readily provide the following algebraic data type; the type parameter of Tree denotes the label type, which is String in the example at hand:

```
data Tree a = Node a [Tree a]
```

3.2 Canonical mappings

The above mappings from and to the universal representation type are canonical mappings. These are mappings that can be defined once and for all for a given class of data models (namely for all arbitrary (algebraic) data types in the example at hand). So let us define the mappings data2tree and tree2data. We employ Haskell's 'Scrap your boilerplate' style of generic programming [62,63]. The mapping from data to trees is concisely defined as follows:

```
-- Haskell 98 + common extensions
-- Tree-alize data
data2tree :: Data a => a -> Tree String
data2tree x = Node
                (showConstr (toConstr x)) -- label
                (gmapQ data2tree x)       -- subtrees
```

As the type clarifies, the function data2tree is a generic function: it is polymorphic in the type to be mapped to the representation type. The definition of the mapping reads as follows. Using the primitive access function toConstr, we retrieve the constructor of the datum, which we turn into the string label of the tree using showConstr. Using the primitive traversal combinator gmapQ, we apply data2tree recursively to all the immediate subterms of the datum at hand, resulting in a list of untyped subtrees.

Here is also the inverse mapping — trees to data:

```
-- De-tree-alize tree
tree2data :: Data a => Tree String -> Maybe a
tree2data (Node l ts) = result
 where result = do
                   con <- readConstr resultType l
                   fromConstrL tree2data con ts
       resultType = (dataTypeOf (fromJust result))
```

The application of readConstr maps the string label into an actual constructor of the type to be populated. To this end, we use reflective information about the data type in question; cf. dataTypeOf. We construct a datum from the constructor by applying tree2data recursively on subtrees. The builder primitive fromConstrL takes a function to recursively build subterms, it also takes a constructor from which to build a term, and it takes the list of subterms in the universal representation.

3.3 Mapping customization

The idea is that a canonical mapping is defined by a sort of generic procedure. Hence, we face an extreme form of a *non-canonical* mapping when the mapping is defined 'in all detail'. For instance, recall the point-to-point mappings in GUI data binding. However, we may also leave the grounds of canonical mappings due to customization. That

is, a canonical default may exist, while the mapping setup is prepared to accommodate 'special cases'.

Let us look into object de-/serialization as a scenario that typically involves customization. A serialized default representation is available for each and every object type; the default can be overridden though by the OO programmer on a per-object-type basis. The following VB.NET fragment illustrates plain OO serialization; an object of type BinaryNode is serialized to an XML file (using a SOAP formatter):

```
' VB.NET 7.0 code
Dim myExp = New BinaryNode("")
' ... further object instantiation omitted ...
Dim s = File.Open("foo.xml", ...)
Dim f = New SoapFormatter
f.Serialize(s,myExp)
s.Close()
```

This direction corresponds to the data2tree function given above, except that we serialize (or tree-alize) to XML this time. An OO class is made fit for (de-)serialization by attaching a custom-attribute Serializable to the class:

```
<Serializable()> Public Class BinaryNode
' ... elided ...
End Class
```

The Serializable attribute tells the serialization library that it is 'allowed' to leverage reflective programming to carry out the mapping from objects to XML (and vice versa) in a canonical fashion. Customization of the canonical default is enabled by the following provisions. One can implement a designated ISerializable interface, and in particular, a GetObjectData method to override the generic reflection-based behavior for the serialization of the object's content:

```
Sub GetObjectData(ByVal info As SerializationInfo, _
                  ByVal context As StreamingContext)
 Implements ISerializable.GetObjectData
  ' Identify data for serialization
  info.AddValue("field1", field1 );
  info.AddValue("field2", field2 );
End Sub
```

3.4 Type- vs. instance-based

A *type-based mapping* is defined as a relationship between the two involved data models, while it is assumed that this type correspondence (more or less) directly implies the actual value transformation for the two data models. By contrast, an *instance-based mapping* expresses value transformations directly. Type-based mappings are less expressive because they are also more abstract and canonical. In return, they make it easier to provide *updateability* (i.e., pushing back target-side value modifications to the source) and *composability* (i.e., performing target queries directly on the source).

A clear-cut example of a type-based mapping is canonical XML data binding where any given XML schema is mapped to a corresponding object model based on fixed rules that only refer to type patterns in XML schemas and object models; cf. Sec. 2.3. For instance, a specific mapping for XML-data binding could involve the following type-based mapping rule:

> *Any global element declaration without attribute declarations, with a sequence group for its content model such that the children of the sequence are local element declarations with distinct element names and nominally specified (as opposed to anonymous) content types is mapped to an object type with the global element name as class name (after name mapping), with fields for the local element declarations such that the local element names serve as field names (after name mapping) and the content types of the elements serve as field types (after type mapping).*

We leave it as an exercise to the reader to attempt a classification with regard to the 'type- vs. instance-based mapping' dichotomy for each of the examples of Sec. 2. (As we will argue shortly, such a classification may be difficult at times.) Let us consider a non-trivial but clear-cut example of an instance-based mapping. Suppose we want to extract a problem-specific XML view on some tables in a relational database. There are these tables: Orders, Employee and Customer; the XML view should look as follows:[9]

```
<Customer CustomerID="ALFKI">
  <Order OrderID="10643" />
  <Order OrderID="10952" />
  <Order OrderID="11011" />
  <Employee LastName="Davolio" />
  <Employee LastName="Leverling" />
</Customer>
 ...
```

The XML view can be derived through a nested SELECT statement on the database. Some annotations like FOR XML AUTO, TYPE clarify that the result indeed should be 'rendered' as XML data rather than a list of queried rows:

```
// SQL with XML extensions of SQL Server 2005
SELECT CustomerID AS "CustomerID",
      (SELECT OrderID AS "OrderID"
       FROM Orders "Order"
       WHERE "Order".CustomerID = Customer.CustomerID
       FOR XML AUTO, TYPE),
      (SELECT DISTINCT LastName AS "LastName"
       FROM Employees Employee
       JOIN Orders "Order" ON "Order".EmployeeID = Employee.EmployeeID
       WHERE Customer.CustomerID = "Order".CustomerID
       FOR XML AUTO, TYPE)
FROM Customers Customer
FOR XML AUTO, TYPE
```

[9] Source: http://msdn.microsoft.com/library/en-us/dnsql90/html/forxml2k5.asp

Let us also look at a less clear-cut example — the generic function for mapping Haskell terms of arbitrary types to trees according to a universal representation type. Do we face an example of a type-based or an instance-based mapping, or is it neither of these?

```
-- Tree-alize data
data2tree :: Data a => a -> Tree String
data2tree x = Node
                (showConstr (toConstr x)) -- label
                (gmapQ data2tree x)       -- subtrees
```

The mapping is type-based because it is fully generic. Likewise, any reflection-based mapping (such as object serialization) counts as type-based. Once customization enters the scene, the mapping may become partially instance-based. Here we assume that customization is regarded as a means of providing type-specific value transformations. The XML view example, given above, is clear-cut instance-based since it defines a query that is fully specific to certain tables, columns thereof and key-value relationships. One could impose a certain XML schema ('type') on the result of the instance-based mapping, or one could even attempt to infer such a schema ('type'), but the mapping is nevertheless defined as a value transformation.

3.5 The programmatic-to-declarative scale

It is common to say about mappings that they are defined *programmatically* or *declaratively*. These are not absolute concepts, but we attempt to justify these terms anyway since they are in common use and they may eventually turn out to be useful — once we better understand the scale; once we are in possession of proper definitions. (This is a future-work item.)

Programmatic mapping We use this term to refer to a mapping that is defined through program code such as in a general purpose programming language or a (typically Turing-complete) data processing language such as SQL, XSLT, XQuery, C#, Haskell, Java or VB. Hence, the FOR XML AUTO example from the previous section would count as programmatic. Let us review another programmatic mapping example. That is, let us map a business-object type, Order, to an XML type, Invoice. The following programmatic mapping code 'dots' into the business object, it eventually reaches sub-objects for customers and addresses, and it assembles new XML objects (using XML data binding) for invoices and items thereof; the encoding uses LINQ's SQL-like query syntax to iterate over object collections [73]:

```
// C# 3.0 / LINQ code (as of May 2006)
public static XmlTypes.Invoice Map(ObjectTypes.Order ord)
{
   return new XmlTypes.Invoice {
      name   = ord.cust.name,
      street = ord.cust.addr.street,
      city   = ord.cust.addr.city,
      zip    = ord.cust.addr.zip,
```

```
      state   = ord.cust.addr.state,
      items   = (from i in ord.items
                  select new XmlTypes.Item {
                      prodid    = i.prod.prodid,
                      price     = i.price,
                      quantity  = i.quantity }).ToList(),
      total   = ord.computeTotal()
    };
}
```

Declarative mapping We use this term to refer to a mapping that is not defined as an immediately executable program with an intrinsic operational semantics by itself; instead the mapping (description) is associated with an operational semantics by means of a separate interpretation, translation or code generation. For instance, the MAPSYM and MAPRULE constructs of Sec. 2.1 suggest that Maptool descriptions amount to declarative mappings. The implicit assumption is that a declarative mapping lends itself 'more easily' to different analyses and interpretations. For instance, one should expect that declarative mappings are amenable to updateability (or reversibility, two-way updates) 'more easily'.

Let us review another declarative mapping example. That is, let us look at object-relational mapping as it is done in HIBERNATE — an approach for relational persistence for 'idiomatic Java'.[10] The following fragment of a *class-centric mapping specification* defines the structure of a class Cat in terms of a table CATS. Class properties are associated with table columns; a generator class is associated with the id column:

```
<class name="Cat" table="CATS">
  <id name="id" column="uid" type="long">
    <generator class="hilo"/>
  </id>
  <property name="birthdate" type="date"/>
  <property name="color" not-null="true"/>
  <property name="sex" not-null="true" update="false"/>
  <property name="weight"/>
  <many-to-one name="mate" column="mate_id"/>
  <set name="kittens">
    <key column="mother_id"/>
      <one-to-many class="Cat"/>
  </set>
</class>
```

This declarative mapping admits the derivation of an actual Java class and a relational schema such that the derived class facilitates the population of objects from relational data and the persistence of objects as relational data.

[10] Source: http://www.hibernate.org/hib_docs/reference/en/html/mapping.html

3.6 Annotations vs. references

The Hibernate mapping of the previous section essentially prescribed both the ultimate Java class and the associated relational schema. In many mapping scenarios, the actual source and/or target types of a mapping predate the mapping effort, in which case the purpose of a (declarative) mapping specification is really just to associate two existing data model(s) with mapping rules or to define a new data model in terms of a given one. In these cases, there exist two major options:

- *Annotations*: A data model is physically annotated with mapping rules.
- *References*: The mapping specification refers to components of the data model(s).

We illustrate this variation point in the context of XML data binding using the JAXB technology for Java [91]. JAXB admits mapping customization using both inline schema annotations and a standalone mapping specification with schema references. Let us pick up the 'shapes example' again, which we started in Sec. 2.3, when we discussed some drawbacks of xsd.exe's canonical mapping. The mapping of JAXB is largely different from xsd.exe's mapping, and so we encounter different issues:

- Recall the choice group of the Widgets element:

  ```
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="Rectangle" type="Rectangle"/>
    <xs:element name="Square" type="Square"/>
    <xs:element name="Circle" type="Circle"/>
  </xs:choice>
  ```

  The canonical default mapping results in the following field:

  ```
  protected List<Object> rectangleOrSquareOrCircle;
  ```

  This mapping mostly illustrates Java's (as much as .NET's) weak story regarding 'old-style' discriminated unions. Still, there is room for improving the mapping result. In particular, we would like to use customization to replace the generated 'ugly' name rectangleOrSquareOrCircle by a more reasonable name, say Shapes.
- As we discussed for the xsd.exe technology, it is limiting that Rectangle, Square, Circle are unrelated base classes. Again, we would like to enable polymorphism by establishing a common base class using customization. As an aside, the .NET technique of using partial classes (cf. Sec. 2.3) cannot be adopted here because Java (1.5) does not offer an equivalent mechanism. (We could use aspect-oriented language extensions though [53].)

We will address both issues by using JAXB's customization mechanisms.[11]

Annotation-based mapping We start the XML schema for shapes all over again. The annotation-based approach requires that we use JAXB-specific annotations in our schema. Hence, we need to bring all namespaces for JAXB into the scope:

[11] Source: http://www.onjava.com/pub/a/onjava/2003/12/10/jaxb.html

```
<!-- XML schema -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
           xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
           jaxb:version="2.0" jaxb:extensionBindingPrefixes="xjc">
<!-- Cont'd below -->
```

Right at the top level of the schema, we attach a new default base class, where we follow the rules of JAXB's schema for such binding declarations, and we also adhere to the rules for placing annotations in an XML schema. That is:

```
<!-- Cont'd from above -->
<xs:annotation>
  <xs:appinfo>
    <jaxb:globalBindings>
      <xjc:superClass name="drawApp.Shape"/>
    </jaxb:globalBindings>
  </xs:appinfo>
</xs:annotation>
<!-- Cont'd below -->
```

We should note that this is a poor man's solution because Shape will now serve as the base class for *all* classes that are derived from the schema at hand. This is acceptable for the particular shapes schema. We also note that subclassing of the generated classes Rectangle etc. will be necessary once implementations of the draw method are to be added. (In the VB.NET version we were able to add the subclass-specific implementations of the draw method retroactively to the generated classes Rectangle etc. with the help of the partial-class technique. Again, we could use Java extensions, such as open-class mechanisms of aspect-oriented programming in similar ways [53].)

The other issue — provision of a reasonable name for the widgets collection — is resolved as follows. We add an annotation to the Widgets element such that the name of the generated Java property is defined as Shapes:

```
<!-- Cont'd from above -->
<xs:element name="Widgets">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:annotation>
        <xs:appinfo>
          <jaxb:property name="Shapes"/>
        </xs:appinfo>
      </xs:annotation>
      <xs:element name="Rectangle" type="Rectangle"/>
      <xs:element name="Square" type="Square"/>
      <xs:element name="Circle" type="Circle"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

We elide the rest of the schema because it does not contain further annotations.

Reference-based mapping JAXB offers a reference-based technique for customization that simply exploits the following facts. First, XML schemas are XML documents. Second, fragments in XML document can be addressed precisely through XPath expressions. It is indeed straightforward to transcribe the earlier annotation-based mapping description such that all binding declarations are gathered separately, and they are attached to schema parts through XPath references. So we reuse the earlier annotations, as is, but we collect them in designated JAXB bindings elements. For brevity, we only show the customization of the choice group:

```
<jxb:bindings node="//xs:element[@name='Widgets']//xs:complexType//xs:choice">
  <jxb:property name="Shapes"/>
</jxb:bindings>
```

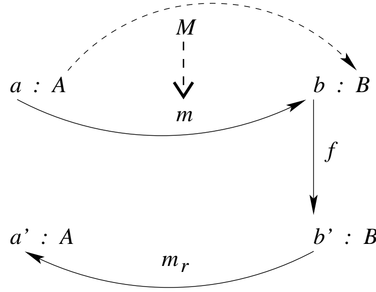
We can see that the XPath expression under node descends into the element declaration of @name='Widgets', then into the complexType component underneath, until it hits on the subtree for the choice. One may argue whether or not the use of a low-level (schema-unaware, very syntactical) selector technique like XPath provides a sufficient level of abstraction. One alternative is offered by XML schema *component designators* [97] — an XML language for identifying XML Schema components.

3.7 Updateability

Various mapping scenarios require updateability in the sense that target-side value modifications must be pushed back to the source. We have discussed this issue already in the specific context of GUI data binding. Object-relational mappings constitute another general class of mappings with an updateability requirement. That is, the database instance and its manifestation as an object graph are supposed to stay in sync. In the source-to-target direction, syncing is potentially just seen as a 'refresh' issue. In the target-to-source direction, syncing may require building the converse of a mapping (a 'view') that was originally thought of as being directed from source to target. For instance, an SQL-like view seems to be directed in that sense. Indeed, the *view-update translation problem* for databases [10,34] is the classic form (and challenge) of an updateable mapping. The subsequent discussion is meant to provide an account on scenarios for updateability, overall attacks, practical challenges and available foundations such as *data refinement* and *bi-directional transformations*.

Alleviated or missing updateability requirement Updateability (or reversibility) is not a universal requirement for mappings. For instance, concrete-to-abstract syntax mappings are mostly not expected to be invertible. However, some degenerated form of updateability (such as origin tracking [22]) may still be required. Consider a language implementation with a type checker that consumes abstract syntax; when type errors are found, the type checker must be able to refer back to the original part of the input — for the programmer's convenience who needs to understand the error message.

'Near-to' bijections Let us consider a basic (restricted) form of updateability:



In this figure, we face types (data models) A and B and instances (elements) a and b. We also indicate the possibility of a type-level mapping M from A to B, but M is not essential. It is essential though that there an instance-level mapping m that maps a to b. Updateability of the mapping means that there is instance-level mapping m_r with which a changed target value, b', can be mapped back to an accordingly changed a'.

It is clear that m_r should be the converse of m. For a bijective m, updateability is trivially implied. However, this assumption is quite restrictive. For instance, a mapping that provides a *view* on a source (just as an SQL view) will be non-injective. In other scenarios, injectivity may be feasible but surjectivity cannot be delivered. For instance, the target types of a mapping may be intrinsically richer (say, more liberal). In the sequel, we identify deviations from bijectivity.

For a non-injective m, we have to come up with a heuristic to resolve the choice points when mapping back b' to a'. Suppose m projects *away* data (such as in a general SQL SELECT statement), we would have to expect that m_r somehow puts back the eliminated data. This is mathematically impossible for the shown diagram. We really need to have access to more information such as the original value a. We get to this different scheme in a second.

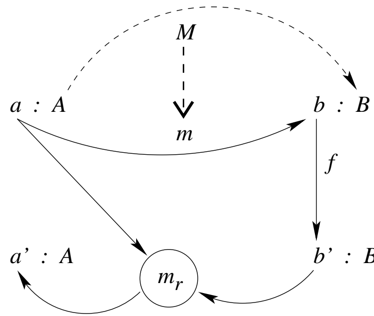
For an injective but non-surjective m, there are essentially two major cases. The first one is that B is representationally richer, but m and modifications on B do not (or are not supposed to) exploit this generality. In this case, we are still able to define a suitable m_r such that the composition of m and m_r is the identity. This case is nicely backed up by research on data refinement [42,74,78,79,6]. (The functions m and m_r are called the 'representation' and 'abstraction' mappings in standard data refinement terminology.) For instance, one can easily see that the following types are in refinement order:

$$X \subset X + 1$$
$$X \to (Y + Z) \subset (X \to Y) \times (X \to Z)$$

The first inequality is the abstract version of mapping a non-nullable type to a nullable type such as mapping a NOT NULL column of a 'value type' to an object type (which is 'nullable' because it is reference type). The second inequality demonstrates the elimination of sums through refinement. Again, the right-hand side admits more values such as a $y \in Y$ and a $z \in Z$ both being associated with the same $x \in X$. However, if the contract is such that the richer representation must not (cannot) be explored, then the mapping is still updateable.

For an injective but non-surjective m, we could also face the case that B is designed to maintain extra data along the life cycle of the mapped data on the target side. In this case, we assume that any b' can be narrowed down to the range of m in a meaning-preserving manner. Here is a simple example: change flags on the target side. These change flags may be essential for the optimized propagation of updates from the target to the source, but they are semantically irrelevant because we could (in theory) assume that all rows were changed.

Facilitation of original value There are several ways to improve on the discussed notion of 'near-to' bijections. Perhaps the must general and fundamental improvement is to take the original value, a, into account when mapping b' to a':



That is, when mapping back a piece of target data, b', we may also observe the associated piece of source data, a. Therefore, m_r can now compensate for a non-injective facet of m. (We may want to pass b to m_r too, or we may assume that m_r 're-evaluates' $m(a)$ in case it needs b.)

The data sets of Microsoft's ADO.NET technology for object-relational mapping instantiate this idea.[12] In-memory rows from the database carry identities (based on real or made-up primary keys). Hence, client-side changes can be pushed back to the database using 'keyed' UPDATE statements.

Bi-directional transformations Pierce, Hu and others have recently developed a formal notion of bi-directional transformations [36,45,13] that provide updateability for mappings on data. This approach facilitates the original value, but its real insight is centered around the discipline of transformation. What they call 'transformation' is (intuitively) a source-to-target instance-level mapping function which however comes with two 'interpretations' *get* and *put* for performing the mapping both ways. The bi-directional transformation literature studies the various primitive transformations and composition operators that can be fitted into this conceptual framework. Initially, this line of work applied to tree data only, but very recent developments also cover relational data. In fact, it had been observed from the very beginnings that bi-directional transformations are quite related to view-update translation for databases [10,34].

[12] Source: `http://msdn2.microsoft.com/en-us/library/y2ad8t9c(VS.80).aspx`

It will be interesting to apply such theory to actual mapping problems such as object-XML mapping or object-relational mapping. We think that it is necessary to study updateability (say, bi-directional transformations) in a context that pays attention to all relevant concerns including these: remoting queries and DML operations, dealing with transient state, making scalar types vs. structured types updateable, making mapping lazy, and so on.

3.8 Usage protocols

A very complex topic that we can only touch upon here is the provision of protocols as a complement of the mere data-modeling aspects of mapping. When talking about mappings, one may easily focus on typing issues and neglect the protocol that goes with the mapped data. An intuitive definition of the term 'usage protocol' is this: a usage protocol describes *order and conditions for the invocation of methods* in a (data access) API.

Protocols in XML data binding For instance, let us consider the protocol for using a schema-derived object model in the context of XML data binding (i.e., object-XML mapping). The simple version of the protocol goes as follows:

1. De-serialize XML document into objects.
2. Operate on bound object structure using plain OO programming.
3. Serialize objects back to XML document.

This list is superficial. Here are some neglected protocol issues:

- Construction of structured content follows a certain protocol.
- Mixed content observation and injection requires special protocols.
- On-demand validation of global or other constraints may be provided.
- The tree semantics of XML imposes a certain contract on DML operations.
- Access to low-level XML views may require on-the fly (de-) serialization.
- There may be a protocol to handle transiently invalid content.

The data-sets protocol Let us consider the usage protocol for multiple-tier architectures using ADO.NET, in particular the ability of changing disconnected data sets that need to be committed later to the database.[13] The protocol identifies the following steps for creating and refreshing a data set, and in turn, updating the original data:

1. Build DataSet.
2. Fill DataSet with data from a data source using a DataAdapter.
3. Change DataSet by adding, updating or deleting DataRow objects.
4. For 2-tier apps:
   (a) Invoke Update on DataAdapter with the above DataSet as an argument.
   (b) Invoke AcceptChanges on DataSet.

[13] Source: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemDataDataSetClassTopic.asp

5. For n-tier apps:
   (a) Invoke GetChanges to create a 2nd DataSet that features only the changes.
   (b) Send the second DataSet to the middle-tier via WebServices.
   (c) On the middle-tier,
   invoke Update on DataAdapter, with the 2nd DataSet instance as an argument.
   (d) From the middle-tier,
   send the updated DataSet back to the client via WebServices.
   (This DataSet may have server generated columns set to the latest value).
   (e) On the client-side,
   invoke Merge on original DataSet to merge the received DataSet, and then
   invoke AcceptChanges on the original DataSet.
6. Alternatively, invoke RejectChanges to cancel the changes.

The general observation is that the definition of a mapping is only complete when protocol issues are clearly defined, too. Unfortunately, in practice, mappings are not rigorously defined in this respect.

3.9 Further reading

There are several fields in software engineering and programming language theory that involve notions of mapping with similarities to mappings in data processing. We do not dive into those fields here, but we document them as related work:

- As mentioned before: data refinement [42,74,78,79,6].
- As mentioned before: bi-directional transformations [36,45,13].
- Consistency maintenance in software modeling [60,46].
- Consistency maintenance in cooperative editing [24,90].
- Data views in functional programming (patterns for ADTs) [98,16,77].
- Program views in intentional programming and fluid AOP [88,5,52].
- Reconcilable model transformation in model-driven development [85,26].
- Source-code modeling in re-/reverse engineering [83,58,69,44,56].
- The general notion of coupled transformations [61].

4 Cross-paradigm impedance mismatches

Having discussed mapping examples and concepts, we still need to get a better handle on the following question: ‘Why is it that inter-paradigmatic mappings are so difficult?’. The present section identifies and illustrates the impedance mismatches amongst the major paradigms for data modeling and processing: Cobol, object, relational and XML. We focus on the technical dimension; we attempt to circumvent the ‘cultural’ dimension of impedance mismatches [7]. (This separation is not always easy to mark off though.)

4.1 Characteristics of major paradigms

The impedance mismatches are rooted in the different characteristics of the paradigms. So we recall these characteristics here — as a means of preparation.

Object-oriented programming

For simplicity, we focus on mainstream, class-based, imperative, typed, object-oriented programming languages like C++, C#, Java, and VB.

- *Reference semantics*. Object structures are graphs that are basically assembled by storing references to objects in data fields of objects. An OO language may prefer not to surface the distinction between objects and object references (say pointers), but the semantics of objects is reference-based anyhow. Object construction returns a reference to the newly constructed object. Objects are passed by reference to methods. Objects can be compared for equality in the sense of object identity (i.e., object reference equality).
- *Encapsulation*. The data part of object structures does not exist in isolation. Instead, an object is a capsule of data *and* behavior (i.e., methods). The interface of an object typically provides restricted access to the data fields (i.e., to the low-level state) of an object. Consistent changes of an object's state (as well as conglomerations of objects) are to be achieved through the behavioral interface of an object. (In object-based languages and advanced frameworks, methods may be part of an object's state, too.)
- *Properties*. Data fields are often not directly exposed through an object's interface, but the object's state is instead published through *properties* adding a level of indirection. This idiom allows one to hide representation details while still providing a structural view on the object's state.
- *Abstract classes and interfaces*. An object model defines its data model potentially also through types that cannot be directly instantiated. That is, OO languages allow for abstract classes, and most typed OO languages support interfaces by now.
- *Subtype polymorphism*. Classes and interfaces are arranged in subtype hierarchies. Each variable is declared with a static type that serves as a bound for the type of objects that may be assigned to the variable. A subtype may add components to the state, and it may enrich the interface. Ideally, subtypes preserve the observable behavior of the supertype; cf. the *substitution principle* [66].
- *Generics*. Most typed OO languages support generics (parametric polymorphism) by now. That is, classes, interfaces and method types can be parameterized in types.

Relational databases

We restrict ourselves to SQL databases in the sequel.

- *Relational algebra*. Conceptually, database tables are mathematical relations in the sense of sets of tuples over *scalar* data [19,21]. One can process these relations in terms of set-theoretic operations (such as union, difference and intersection) as well as relational algebra operations (such as projection, selection, Cartesian product and join). In practice, we deviate from this mathematical ideal a bit, e.g., order of rows does matter in tables; SQL's SELECT statement combines several operations.

- *Keys*. Both at a fundamental level (i.e., relational algebra) and in practice, table columns may specifically serve for the global identification of table rows (cf. primary key) such that other tables may refer to the identified rows (cf. foreign key). It is a crucial ingredient of relational schema design to identify such primary and foreign keys as they will be used in queries (for joins) and in ensuring the referential integrity of the database as a whole.
- *Data integrity*. More generally, a database schema makes contributions to the effective maintenance of data integrity. To this end, integrity constraints (such as foreign key constraints), cascading operations and triggers can be used.
- *Transactions*. DML operations on a database are scoped in groups that are called transactions. Only the successful completion of such groups leads to an observable state change of the database. Transactions facilitate consistency in databases. For instance, the insertion or deletion of a row in one table may only be valid in combination with updating rows in other tables. Transactions may be expressed as stored SQL procedures that consist of SQL DML statements.
- *Schema evolution*. Within limits, a database schema can be adapted, perhaps even while the database is on-line. The schema may evolve in such a way that all or most previously valid queries continue to be valid.
- *Views*. In addition to physical tables, there can be views, which are defined by SQL queries. Views are never materialized; the defining queries are executed once the view itself is queried. Updates on views are relatively restricted.

XML document processing

We have in mind XML processing using XPath, XSLT, DOM, XQuery and friends. We also include uses such that XPath or other XML languages are embedded into general-purpose languages such as Java or C#.

- *Tree structure*. XML elements are normally organized as trees — as opposed to flat tuples or arbitrary graphs. One may use IDREFs to refer to remote elements, but this idiom is only occasionally used in practice. Also, XML *types* may be recursive, but the prevailing concept in XML is hierarchical, tree-like organization of data.
- *Element vs. attribute dichotomy*. “A perennial question arising in the mind [... of XML designers ...] is whether to model and encode certain information using an element, or alternatively, using an attribute. ... Experienced markup-language experts offer different opinions ...”[14]
- *Mixed content*. One may mix structured content (elements) and text. Processing instructions (PIs) and comments may occur, too. Some XML use cases require high fidelity: all details of text, PIs and comments are to be preserved. This issue is very similar to layout and comment preservation for programming language processing [93,56] (also known as syntax retention).

[14] Source: http://xml.coverpages.org/elementsAndAttrs.html

– *Order matters*. Element tags are not meant to be (unambiguous) selectors or labels. Multiple element particles with the same element name may occur in a content model. XML processing functionality may care about the order of elements. Order also matters with regard to querying. That is, queries (based on XPath and friends) are normally supposed to return elements in *document order*.
– *The XML infoset*. The representation-biased view on XML is complemented by a more abstract interpretation of XML documents: the *infoset* [95]. This semantic domain regulates what sort of information is associated with each node in a well-formed (and not necessarily valid) XML document. The infoset hints at the *axis-based navigation* style for XML. That is, one can navigate to the children, to the parent, and to the siblings. In fact, the document object model (DOM [94]) almost directly implements the infoset semantics. In reality, there is not just a simple data model for XML. Most notably, each XML API implements a slightly different variation on the infoset. Also, the data model of XPath is yet again slightly different from plain infoset.

Cobol

Cobol is not just the most widely used programming language for data-processing applications; it is in fact a language that has been designed to specifically serve this role. More than that, Cobol continuously evolves to co-exist with other paradigms. (For instance, Cobol has been turned into a proper OO language over the last decade or so [48]. Admittedly, OO Cobol sees limited adoption.) Here are Cobol's characteristics:

– *Files as a language concept*. Unlike development platforms of the last 10 years or so, persistent data processing is not viewed as an 'API issue' in Cobol. Instead, statically typed language constructs for keyed and sequential file access amount to an intrinsic component of the language since the 1960s. ("No strings".) The concept of keyed files is similar to the relational model except that file access is record-based and general joins need to be rolled out as nested loops.
– *Database support*. Embedded SQL (optionally combined with transaction monitors like CICS) allows for processing relational data in Cobol code. Embedded SQL can be pre-compiled (including compile-time data dictionary access), which implies static typing and enables optimizations. SQL queries are executed using a cursor model, and result rows are stored in accordingly structured (potentially generated) group fields, based on a simple mapping of SQL data types to Cobol types.
– *XML support*. Conceptually, records (and Cobol data in general) are described through arbitrarily nested group fields. This is already a good fit with the tree-like organization in XML except for the issue of unbounded occurrence constraints and choice types. Also, Cobol readily offers 'representation-oriented' data types, just as XML schema. Additional native XML support is being added to the standard [8]. This addition allows file processing on XML data and validation. Content models can be described more or less like normal file records. A cursor-based model resolves the issue of unbounded occurrence constraints. Choices may be treated procedurally by tag inspection.

4.2 An open-ended list of mapping issues

We attempt to pinpoint ‘issues’ that witness impedance mismatches for inter-paradigm mappings. The issues are phrased as questions. We reckon that each such question does not lend itself to a trivial, unambiguous and non-debated answer.

Map a relational schema to an object model

Such a mapping is needed when an OO application requires access to business data that happens to reside in a relational database. This is perhaps the most common mapping scenario in IT today up to a point that experts have labeled this problem as the ‘Vietnam of Computer Science’. There are various more or less complex technologies in existence that attempt to address this problem, e.g., EJB and Hibernate, and some technologies were never completed. We phrase some issues as questions:

- How to map database schemas to class hierarchies?
  (What data is going to be private if any? What to use inheritance for, if at all?)
- How to perform queries on objects (that represent relations)?
  (Should we mimic SQL? Should we use OOQL, XPath or XQuery?)
- How do foreign key constraints show up in the behavior of objects?
  (How to map foreign key constraints to an OO design?)
- How would we possibly carry out schema evolution on the OO program?
  (Also, can we achieve independent evolution of database schema and object model?)
- How to map SQL views and stored SQL procedures to objects?
- How to enable transactions in the object-life cycle?
- Can we make any use of interface polymorphism?
- How to map object access to SQL queries?

Map an object model to a relational schema

Such a mapping is needed in the following situations: (i) the database is meant to serve for plain object persistence; (ii) the architecture of an OO application is constrained to expose its object model as a relational schema, which may be considered as a strong version of (i). In both cases, the mere mapping problem might couple up with a migration problem. That is, we may need to re-engineer the OO application that pre-dated the mapping requirement. ‘Normal objects’ are to be replaced or complemented by database-access objects.

- What classes, fields and properties are mapped to relations?
  (In particular, what private fields have to be persisted if any?)
- How do we map single/multiple OO inheritance to relations?
  (How does this affect querying tables that correspond to subclasses?)
- How to extract foreign key constraints from OO designs?
- How to extract NOT NULL constraints from OO designs?
- How to map Eiffel’s, Java’s or .NET’s generics to the database?
- What to do about interface polymorphism, if anything?

Map an object model to an XML schema

Such a mapping is needed in the following situations: (i) data import and export for the sake of open, interoperable software applications; (ii) XML-based persistence; (iii) remote-method invocation and web services. Object models seem to lend themselves to reasonably restricted XML schemas. However:

- What classes, fields and properties need to be mapped anyhow?
  (In particular, what private fields are part of the intrinsic object state?)
- How do we draft the hierarchical organization of the XML data?
  (What associations to represent through hierarchy? Should we use IDREFs?)
- What to do about sharing or cycles in the object graph?
  (How do we even know for sure where sharing and cycles may occur?)
- How to enable platform interoperability (cf. Java vs. .NET)?
- What to do about interface polymorphism, if anything?
- Which XSD organization style to use when?
- How to map generics to XML schema?

Map an XML schema to an object model

That is, the XML schema serves the role of a 'first-contract' data model in this case. The overall scenarios for this mapping direction are more or less the same as for the other direction: objects to XML, but this time we face the full generality of XML schemas as opposed to the subset that is targeted by a given 'objects to XML' approach.

- How to group tree elements in objects?
- How to provide fidelity for mixed content?
- How to map ID literals into object references?
- Do simple XSD types constitute wrapper classes?
- How to map identity constraints to OO behavior?
- How to map facets (maxInclusive etc.) to OO methods?
- How to represent order constraints in OO code if at all?
- How to map type derivation by restriction to OO mechanisms?
- How do we cope with XML data that does not comply to the schema?
- Do we need to distinguish elements from complex types in OO types?
- How to map anonymous model groups to fields/properties in OO code?
- Can we enable independent evolution of XML schema and object model?

Map an XML schema to a relational schema

Such a mapping is needed when we want to use a relational database as an XML store. (In practice, we even may want to store untyped XML data or to neglect the XML schema for the purpose of storing XML data.) Another scenario for XML-to-relational is that we actually aim at a faithful relational schema so that we can operate on the data in two worlds: XML and SQL.

- How to 'normalize' the XML schema?
- How to map XSD's built-in simple types to SQL data types?
- How to avoid clashes of XML IDs from different documents in the database?
- How to (efficiently) support XPath et al. on the relational image?

Map a relational schema to an XML schema

Such a mapping is needed for the provision of an XML view on relational data. As in the case of several previous mapping couples, we may have a choice between canonical mappings (that take any relational schema and expose it as XML without any contribution from a programmer) or custom mappings, where the programmer specifically describes the shape and the content of the desired XML view relative to the relational schema. (These two options are sometimes also referred to as *prescriptive* vs. *descriptive* mappings.)

- When to use IDs/IDREFs and when to use nesting?
- How to deal with circular reference chains in tables?
- How to map SQL data types to XSD's built-in simple types?
- *Exercise to the reader: find some more variation points.*

4.3 Exemplar frictions

We increase the level of detail by discussing exemplar frictions.

OO lacks foreign key constraints

Foreign key constraints in relational databases serve foremost for referential integrity in a database. One cannot accidentally delete a master row if there are still references to this row from elsewhere through foreign keys. Modern databases (such as SQL 92 variants) provide support for *cascading deletes and updates* such that update or delete operations are distributed automatically from the table with the primary key to tables with corresponding foreign keys. (In addition, there is also the *trigger* technique to achieve this behavior with slightly more effort.)

An example follows. Let us assume a stock table that contains a list of items that a shop stocks and sells, as well as a stock transaction table that contains a list of purchases and sales for each stock item. We can only delete a stock item if there are no transactions left that refer to the stock item. However, a cascading delete makes sense here: the deletion of the stock item should imply the deletion of the transactions for this stock item. This is expressed by the ON DELETE CASCADE phrase as part of the foreign key constraint.

```
// SQL Server 2000 code
CREATE TABLE stock_trans
(
trans_id  int  NOT NULL IDENTITY PRIMARY KEY,
stock_id  int  NOT NULL REFERENCES stock(stock_id) ON DELETE CASCADE,
//  ...  further  columns  elided  ...
)
```

Cascading operations are not readily available in the OO paradigm. Let us assume that there are classes for stock items and transactions. Each transaction object holds a reference to the corresponding stock-item object. Also, let us assume that we maintain

a collection of stock items. The trouble is that there is no primitive OO operation for the effective eradication of a stock item including all objects that refer to it. One may employ a range of techniques for the encoding of cascading deletes: weak references, explicit memory management with bi-directional references, the publish-subscriber design pattern, designated design patterns [76,75], and ownership types [18,14,17] (which would call for language extensions).

Cobol's REDEFINES

We are asked to migrate the file-based data management layer of an existing Cobol application to database technology. Such a migration consists of three parts: (i) reverse engineering of the file-based data model with the goal to derive a reasonable relational schema; (ii) data conversion to populate the database; (iii) re-engineering of the Cobol code to perform database access in place of file access. We will focus on (i) because we want to illustrate an impedance mismatch between keyed or sequential Cobol files and the relational model. Once we understand the data-model mapping, the actual data conversion (i.e., ii) is relatively straightforward. (iii) is rather involved. Clearly, migration is not the only option. There are cases in which a Cobol system reaches the end of its conceded life, and we are requested to convert the legacy data to a database. In this case, we can ignore (iii).

Cobol offers (unsafe) variants through its REDEFINES clause: a given record can assume different types. The reverse engineering part needs to identify such records and eradicate them in some way. (The relational model does not comprise designated expressiveness for variant records.) Let us consider an example. The following record description for orders distinguishes *header* records vs. *position* records, which both start with a common structure for key data:

```
* Cobol '85 code
 FILE SECTION.
  FD    ORDER-FILE.
  01    ORDER-RECORD.
* The level 05 group item holds common key data.
    05  ORDER-KEY-DATA.
     10 ORDER-NUMBER      PIC 9(8).
     10 ORDER-ACCOUNT     PIC 9(10).
     10 ORDER-PRODUCT     PIC 9(8).
     10 ORDER-POSNR       PIC 9(4).

* Note:
*  - ORDER-HEADER-DATA is redefined by ORDER-POSITION-DATA
*  - If ORDER-POSNR <= 9, then we face a HEADER.
*  - If ORDER-POSNR > 9, then we face a POSITION.
  05  ORDER-HEADER-DATA.
    10  ... details of header elided ...

  05  ORDER-POSITION-DATA
      REDEFINES ORDER-HEADER-DATA.
    10  ... details of position elided ...
```

The comment reveals that the condition ORDER−POSNR > 9 is supposed to hold for position records. Here, ORDER−POSNR is a data item that contributes to the key of any ORDER record. We may not always find such a helpful comment, neither do we necessarily trust such documentation. Ultimately, we need to engage in reverse engineering indeed. The following code pattern supports the claim in the comment; it aims to read position records while it initializes ORDER−POSNR with 10:

```
* MOVE ALL POSITION RECORDS TO FORM FOR DISPLAY
 INITIALIZE ORDER−RECORD.
 MOVE FORM−ORDER     TO ORDER−NUMBER.
 MOVE FORM−ACCOUNT TO ORDER−ACCOUNT.
 MOVE FORM−PRODUCT TO ORDER−PRODUCT.
 MOVE 10                    TO ORDER−POSNR.
 START ORDER−FILE KEY IS >= ORDER−KEY−1.
 READ ORDER−FILE NEXT RECORD.
 PERFORM WITH TEST BEFORE UNTIL NOT FILE−STATUS−OK
   MOVE CORRESPONDING
     ORDER−POSITION−DATA TO FORM−FOR−ORDER
   READ ORDER−FILE NEXT RECORD
 END−PERFORM.
```

Based on such evidence, we define two database tables corresponding to the variants:

```
// SQL Server 2000 code
CREATE TABLE Order_Header
(
  Order_Number int NOT NULL IDENTITY PRIMARY KEY,
  Order_Account int NOT NULL,
  Order_Product int,
  // ... other data items elided ...
)
CREATE TABLE Order_Position
(
  Order_Number int NOT NULL
   REFERENCES Order_Header(Order_Number),
  Order_Posnr int NOT NULL,
  // ... other data items elided ...
)
```

That is, the order number is designed to be the primary key of the table for header records, while it is a foreign key in the table for position records. The position number only shows up in the table for position records. All general key data (account, product, ...) is centralized in the table for header records. The illustrated mapping requires relatively deep insight, when done manually, or very much advanced program analyses, if the mapping should be automated.

5 Call to arms

For *some* of the above mapping couples, substantial research work has been delivered. For instance, the couple 'map a relational schema to an XML schema' has received ample interest [86,32,27]. For *all* of the above mapping couples, actual technologies do exist and serve business-critical roles everywhere in IT. For *most* of the above mapping couples, the scientific understanding is largely unsatisfactory. Progress is mainly achieved through industrial drive. No integrated foundation of mapping is available. When we compare the situation in mapping with the one in compiler construction, we are clearly in need of a 'Dragon Book' [3] for mapping. However, before someone can write this book, more research is needed. Also, previous and current mapping projects should be carefully analyzed so that the observations (often failures) can be effectively used in new mapping projects and as driving forces for mapping research.

5.1 Overall goals

Foundations — we are in need of general and scalable foundations across paradigms;
Robustness — data access in data processing must not 'go wrong';
Evolvability — data models, APIs and code must not resist change;
Productivity — we need a simpler way of developing data-access layers.

5.2 A list of challenges

Here is a list of challenges that we see ahead of us. We reckon that progress in the area of data-processing application development boils down to progress regarding these challenges.

– *Data models as contracts*. While data-modeling languages such as OCL (UML) and Schematron provide rich *constraint mechanisms*, the transcription of such models to program types may not transport all constraints to the static type system of the programming language at hand. Extending the (static) type system of languages is one direction [72,71,12]. Delivering a language design with support for general pre- and post-conditions is another direction. In fact, a blend of verification, static typing, soft typing [25,101] and dynamic typing seems to be most promising. We quote from [29]:

  > *"We believe that the development of an assertion system [...] serves two purposes. On one hand, the system has strong practical potential because existing type systems simply cannot express many assertions that programmers would like to state. On the other hand, an inspection of a large base of invariants may provide inspiration for the direction of practical future type system research."*

  Further work on the tension between conservative (static) type-system extensions vs. more flexible language support for typing and verification should also benefit from the body of work on OO specification languages such as VDM++; cf. [30].

– *Data-model evolution*. Research on database schema evolution has a history of about 20 years [11,65]. In practice, simple forms of relational schema evolution are established; think of ALTER TABLE in SQL. Related foundations and techniques have been contributed by the field of data re-engineering and reverse engineering [37,67,39] with focus on relational schemas and ER models in information systems. The problem of XML schema evolution may just gain momentum. In OO programming and design, several refactorings address object-model evolution — in particular class refactorings [80,31]. In reality, data-model evolution is a complex topic, and existing methods are difficult to apply (because of restrictions and engineering reasons).
– *Co-evolution of programs*. Assuming that we master the evolution problem for data models themselves, we get to the next hurdle: co-evolution of data-processing programs. This is a particularly challenging (and potentially beneficial) mode of evolution. We need *cross-paradigm transformations* to push changes of relational and XML schemas into application programs, 4GL code, and others.
– *Loose coupling of data models*. Rather than thinking in terms of the propagation of transformations from the data model to the data-model-dependent code, we may also anticipate the problem up-front, and employ an architecture with loose coupling. That is, most business logic would be coded against a more stable object model, which is intelligently mapped to a less stable 'external' data model. Loose coupling would not just help with localizing impact of evolution, it would also allow us to use the preferred internal object model, which may differ from a potentially suboptimal, external data model. Unfortunately, we lack comprehensive foundations of loose coupling.
– *Data model reverse engineering* When we talked about mapping so far, we mostly focused on the technical provision of the mapping assuming that we have a reasonable understanding of the data source and its conceptual, logical as well as physical data model. In practice, we also need to address the problems of 'data-model rot' or 'data-model legacy'. To this end, we need to engage in data-model reverse engineering and re-engineering. These activities may concern both external data models (such as relational schemas) and object models.

5.3 Challenges in detail

We pick out two of the above items for a detailed discussion.

Re-/Reverse engineering of data models

Defining mappings on existing data (or object) models in a concise and robust manner is one issue, knowing the concepts *to map from and to* is another issue that easily dominates the picture whenever we face complex data models, whenever we specifically want to provide a simple-to-digest view.

Let us consider an example. The emerging *Mendocino* project (an effort in which SAP and Microsoft Corp. participate) is aimed at the integration of SAP processes (such

as time management, budget monitoring, organizational management and travel and expense management) directly into Microsoft Office.[15] We may ask how difficult it is to provide such interoperability. One thing to notice is that interoperability is more than a technical term. For Mendocino to be *useful* in the context of office-ware integration, the business processes need to be exposed through *lean data models and APIs*.

In [89], the implementation of SAP R/3 is analyzed in several respects. The published data points lead us to the conclusion that a useful interoperation of SAP with Office is very challenging. We quote some details. For the record, SAP R/3 is implemented in the 4th generation language ABAP/4 (short for Advanced Business Application Programming). According to the results of the study, SAP R/3 consists of 40,000 programs, 34,000 functional modules, 11,500 tables. Regarding the internal data model, it was found that 69% of all data type declarations were not reused (i.e., they were just declared and used *once*); 6.2% were not used at all. For most of the remaining declarations, the number of reuses (in this huge system) is surprisingly small. For four fifths of the reused declarations, reuse was restricted to 2–5 times.

These figures indicate that any reasonable SAP API or any data model for SAP integration would need to make a major effort in order to hide the complexity of the 'as-implemented' data model and to furnish a concise and clear data model that can actually be used by programmers. (The Mendocino project does not start from zero because it can leverage previous interoperability efforts that have gone into the SAP software.)

In the context of software re-/reverse engineering, effective and well-founded methods for data re-/reverse engineering [37,4,43,39] have been developed. We wonder whether these methods can also be adopted in a mapping context that aims at the delivery of programming APIs. In this context, we are not interested in the extraction of relational schemas or ER models for the sake of understanding, system modification or new development; instead we are interested in the provision of programming-enabled views on as-implemented data models.

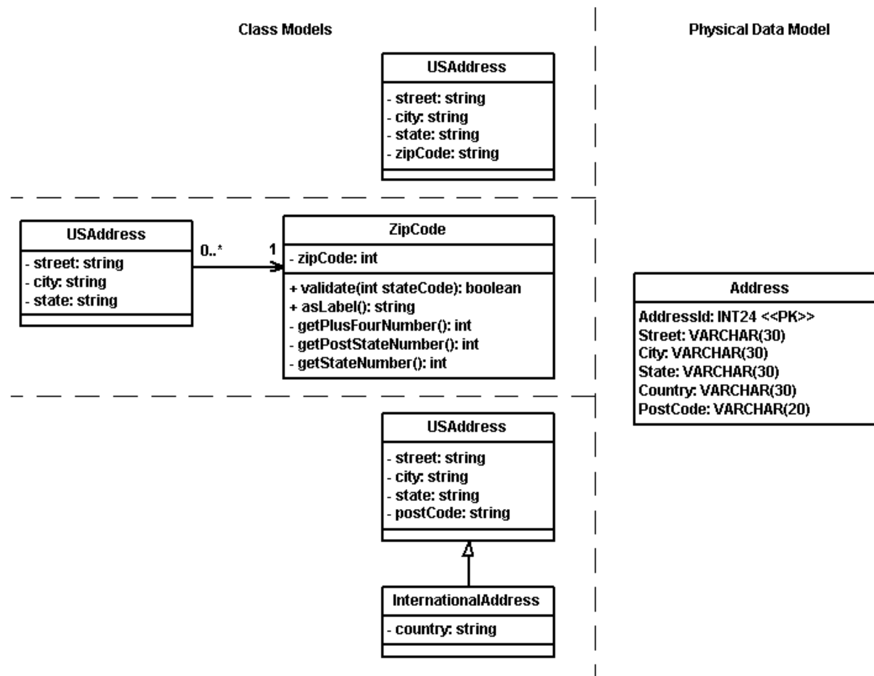
Loose coupling for data and object models

In Fig. 4, we illustrate the not so obvious point that a given data model could correspond to quite different object models in an application. The first object model is 'flat' — just like the physical model. The second object model separates out zip-code objects and extra information about them. Both the first and the second model only deal with US addresses; they omit the country code. The third model defines a flat address structure, but it uses subclassing (in a somewhat pragmatic way) to enable the representation of international addresses.

Differences between external and internal models may arise for various reasons:

- We changed the external data model, but did not change the object model.
- We want to bind to the external data model but favor a different object model.
- We face a legacy object model to be bound to a new external data model.

[15] http://www.sap.com/company/press/press.epx?pressID=4520



(Source: http://www.agiledata.org/essays/drivingForces.html)

Fig. 4. Different object models for the same physical model

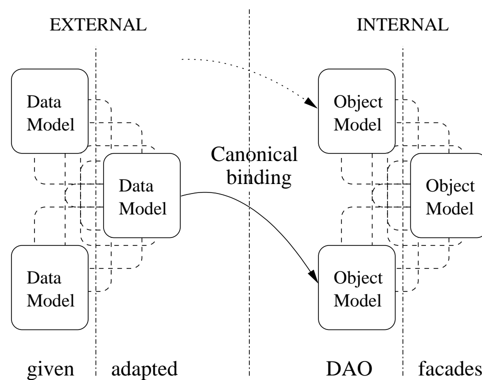
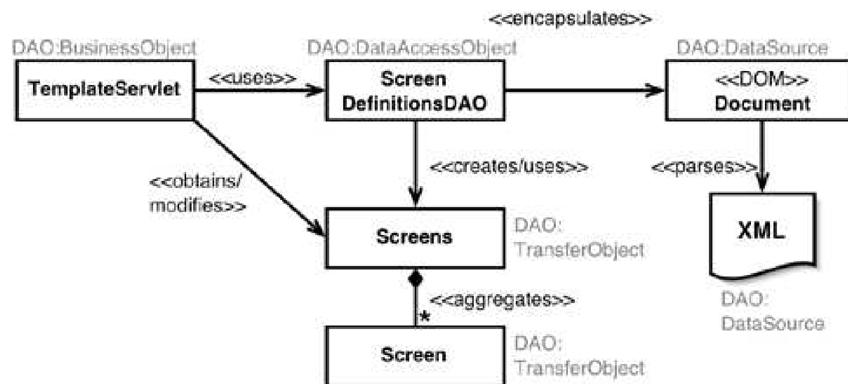


Fig. 5. A mapping web for data-processing applications

Mapping techniques should help us to realize different internal (and even external) models at a high level of abstraction. Unfortunately, in practice, the various models in an application are typically hand-crafted and laborious low-level mapping code is needed to move data back and forth.

In Fig. 5, we sketch the idea of a flexible architecture for data-processing applications. One assumption is that external data models can be combined and transformed



(Source: http://java.sun.com/blueprints/patterns/DAO.html)

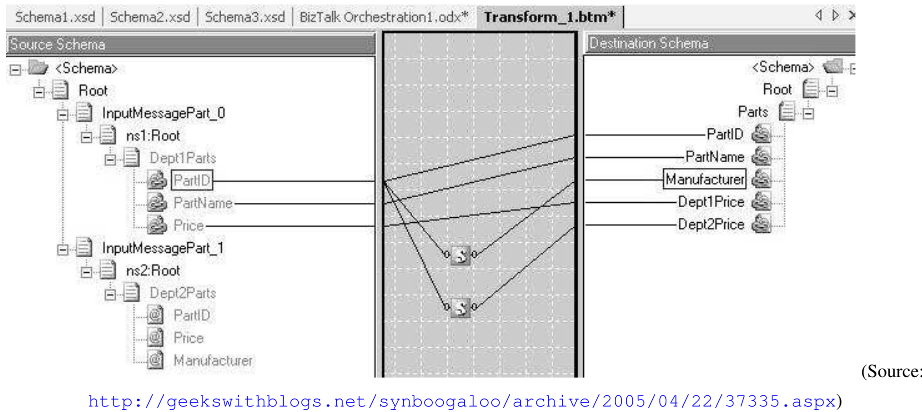
Fig. 6. The design pattern for 'data-access objects' — instantiated for the case of a XML-based data for screen definitions. The business object accesses screen definitions through the data-access object without commitment to the general or particular XML format used underneath.

before entering the software application as canonically derived object models for *data-access objects* (DAO).[16] (The intent of DAOs is the provision of a data-access layer that does not expose implementation details of the underlying data management; see Fig. 6 for an example.) Another assumption is that any number of canonical object models, in turn, can be combined and transformed into 'facades' — these are the object models that are considered useful for implementing the business logic. We may want to assume that the architecture makes it unnecessary to materialize data in interim data-model layers.

For mappings on external models, one can readily use techniques that are specific to the underlying data-modeling paradigm, e.g., XQuery or XSLT for XML, or SQL views for relational databases. (Creating new data models with contributions from different paradigms is more involved.) In Fig. 7, we illustrate tool support for XML schema mapping. (The actual example encounters schema composition by instance-level join.) The tool at hand generates XSLT scripts from the visually designed mapping. Under certain preconditions, one obtains scripts that map both ways.

The situation for mappings on internal models is in flux. There are various design patterns that could be said to help (somewhat) with the design and implementation of mappings: composite, facade, bridge, factory, model-view controller, and most notably, mediator [59,33]. The *mediator* pattern directly allows for the systematic translation of an API into another API (APIs) using connectors as the primary concept. (Notice how well this corresponds to the XML schema mapping exercise in Fig. 7.) The overall notion of mediator is of profound use in the related field of data integration [82,84], but the design pattern is still too weak to do the heavy lifting for mappings in data-processing applications. The mediator pattern does not (nor does any other pattern we know of) provide higher-level operations on data or object models. We can describe the sought-after improvement:

[16] http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
http://java.sun.com/blueprints/patterns/DAO.html



(Source: http://geekswithblogs.net/synboogaloo/archive/2005/04/22/37335.aspx)

Fig. 7. Join two XML schemas (based on the Biztalk technology)

> *The design and the implementation of programmatic object-model-to-object-model mappings is normally carried out at a low level of abstraction, in terms of free-wheeling, basic OO code. We need higher-level language concepts and programming techniques and APIs that enable these mappings more directly.*

Language-integrated query mechanisms (such as LINQ [73]) may serve this agenda.

6 Concluding remarks

We have compiled a survey on mapping practices and mapping issues for data modeling and data-processing with Cobol, object, relational and XML. We have provided rich literature and on-line references. However, the problem is that the intricacies of intra- and inter-paradigm mappings are not fully appreciated by the archetypal research agenda on programming languages and software engineering. As a result, this important field of computer science ends up being driven by industry — not surprisingly more or less in an ad-hoc fashion.

We too adopt an ad-hoc method here to make our point — ‘Google’ science:

- http://www.google.com/search?q=object-relational+mapping
- http://www.google.com/search?q=XML+data+binding
- http://www.google.com/search?q=model-driven+transformation
- http://www.google.com/search?q=aspect-oriented+programming

At the time of writing this conclusion, we observe the following situation. The Google results for the first two links on *object-relational mapping* and *XML data binding* do not lead to *any* research content on the first two pages. (We didn’t continue beyond that.) The Google results for the last two links on *model-driven transformation* and *aspect-oriented programming* readily list several research papers and research projects on the first page. We reckon that, in principle, mappings are worth the same scale of attention in research. Given the fact that IT industry is fighting with various impedance

mismatches and data-model evolution problems for decades, it seems to be safe to start a research career that specifically addresses these problems.

One could perhaps think that the bulk of impedance mismatches will be resolved by language extensions soon. (Why not earlier?) The fix-point of this argument is that we end up with a language in which all mainstream data-modeling paradigms and programming paradigms are 'happily' united. We may need to try indeed, just to see whether the resulting paradigm soup is still digestible. However, the fix-point may be hard to reach anyway. New data modeling and data processing ideas come up all the time. Also, platform providers as much as compiler, IDE, API and tool vendors use differentiation as an intrinsic element of their business strategies. The increasing use of standards in IT (think of reference schemas etc.) is a good thing, but the increasing number of standards (and their size) challenges fix-point iteration, too. So it is essential to continuously cope with diversity. It is therefore a good idea to intensify research efforts on mapping problems that concern Create-Read-Update-Delete applications.

Acknowledgments We are grateful for the insightful proposals by the GTTSE referees and for the mapping discussions with members of the Data Programmability Team at Microsoft. In particular we want to acknowledge contributions by Avner Aharoni, Brian Beckmann, Kawarjit Bedi, Dan Doris, Sergey Dubinets, Charlie Heinemann, Priya Lakshminarayanan, Chris Lovett, Michael Rys, Soumitra Sengupta, Dave Remy and Adam Wiener.

References

1. P. Achten, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Compositional Model-Views with Generic Graphical User Interfaces. In B. Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, volume 3057 of *LNCS*, pages 39–55. Springer, 2004.
2. P. Achten, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Generic Graphical User Interfaces. In P. W. Trinder, G. Michaelson, and R. Pena, editors, *Implementation of Functional Languages, 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003, Revised Papers*, volume 3145 of *LNCS*, pages 152–167. Springer, 2004.
3. A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.
4. P. Aiken, A. H. Muntz, and R. Richards. Dod legacy systems: Reverse engineering data requirements. *Communications of the ACM*, 37(5):26–41, 1994.
5. W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional programming. In P. Devanbu and J. Poulin, editors, *Proceedings, Fifth International Conference on Software Reuse*, pages 114–123. IEEE Computer Society Press, 1998.
6. T. L. Alves, P. F. Silva, J. Visser, and J. N. Oliveira. Strategic Term Rewriting and Its Application to a VDMSL to SQL Conversion. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *LNCS*, pages 399–414. Springer, 2005.
7. S. W. Ambler. The Object-Relational Impedance Mismatch, 2002–2005. Amysoft Inc.; online article http://www.agiledata.org/essays/impedanceMismatch.html.

8. ANSI. Information Technology — Programming languages, their environments and system software interfaces — Native COBOL Syntax for XML Support, Feb. 2005. J4/05-0049, WG4n0229, ISO/IEC JTC 1/SC 22/WG4, ISO/IEC TR 24716:200x(E).
9. J. Aycock. Extending Old Compiler Tools with Meta-Tools. In H. R. Arabnia and H. Reza, editors, *Proceedings of the International Conference on Software Engineering Research and Practice, SERP '04, June 21-24, 2004, Las Vegas, Nevada, USA, Volume 2*, pages 841–845. CSREA Press, 2004.
10. F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.
11. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD International conference on management of data*, pages 311–322, New York, NY, USA, 1987. ACM Press.
12. G. M. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in Cω. In A. P. Black, editor, *ECOOP*, volume 3586 of *LNCS*, pages 287–311. Springer, 2005.
13. A. Bohannon, J. A. Vaughan, and B. C. Pierce. Relational Lenses: A Language for Updateable Views. In *Principles of Database Systems (PODS)*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.
14. C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on object-oriented programing, systems, languages, and applications*, pages 403–417, New York, NY, USA, 2003. ACM Press.
15. A. Brookes. XML data binding. *Dr. Dobb's Journal of Software Tools*, 28(3):26, 28, 30, 32, 35–36, Mar. 2003.
16. F. Burton and R. Cameron. Pattern Matching with Abstract Data Types. *Journal of Functional Programming*, 3(2):171–190, 1993.
17. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 292–310, New York, NY, USA, 2002. ACM Press.
18. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 48–64, New York, NY, USA, 1998. ACM Press.
19. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970. Also published in/as: 'Readings in Database Systems', M. Stonebraker, Morgan-Kaufmann, 1988, pp. 5–15.
20. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
21. C. J. Date. A formal definition of the relational model. *SIGMOD Rec.*, 13(1):18–29, 1982.
22. A. van Deursen, P. Klint, and F. Tip. Origin Tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.
23. U. Eisenecker and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
24. C. Ellis, S. Gibbs, and G. Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.
25. M. Fagan. *Soft typing: an approach to type checking for dynamically typed languages*. PhD thesis, Rice University, 1991.

26. J.-M. Favre. Meta-models and Models Co-Evolution in the 3D Software Space. In *Proceedings of International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA 2003)*, 2003.
27. M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. Silkroute: A framework for publishing relational data in xml. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.
28. J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
29. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM Press.
30. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-Oriented Systems*. Springer, 2005.
31. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
32. J. E. Funderburk, S. Malaika, and B. Reinwald. XML programming with SQL/XML and XQuery. *IBM Systems Journal*, 41(4):642–665, 2002.
33. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
34. G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.
35. R. Gray, V. Heuring, S. Levi, A. Sloane, and W. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, Feb. 1992.
36. M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical Report MS-CIS-03-08, University of Pennsylvania, 2003. Revised April 2004.
37. J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Schema Transformation Techniques for Database Reverse Engineering. In *Proceedings, 12th Int. Conf. on ER Approach*, Arlington-Dallas, 1993. E/R Institute.
38. J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the 17th ACM conference on object-oriented programming, systems, languages, and applications, OOPSLA'02*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, Nov. 4–8 2002. ACM Press.
39. J. Henrad, J.-M. Hick, P. Thiran, and J.-L. Hainaut. Strategies for Data Reengineering. In *Proceedings, Working Conference on Reverse Engineering (WCRE'02)*, pages 211–220. IEEE Computer Society Press, Nov. 2002.
40. A. Herranz and P. Nogueira. More Than Parsing. In F. J. L. Fraguas, editor, *Spanish V Conference on Programming and Languages (PROLE 2005)*, pages 193–202. Thomson-Paraninfo, 14–16 September 2005.
41. R. Hirschfeld and R. Lämmel. Reflective Designs. *IEE Proceedings Software*, 152(1):38–51, Feb. 2005. Special Issue on Reusable Software Libraries.
42. C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatic*, 1:271–281, 1972.
43. K. Hogshead Davis and P. Aiken. Data Reverse Engineering: A Historical Survey. In *Working Conference on Reverse Engineering, WCRE 2000*, pages 70–78. IEEE Computer Society Press, 2000.
44. R. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171. IEEE Computer Society Press, Nov. 2000.

45. Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189. ACM Press, 2004.
46. Z. Huzar, L. Kuzniarz, G. Reggio, J. Sourrouille, and M. Staron. Consistency Problems in UML-based Software Development II, 2003. Workshop proceedings; Research Report 2003:06.
47. ISO. ISO/IEC 14977:1996(E), Information technology — Syntactic metalanguage — Extended BNF, 1996. International Organization for Standardization.
48. ISO/IEC. Information technology — Programming languages — COBOL, 2002. Reference number ISO/IEC 1989:2002(E).
49. S. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.
50. M. de Jonge and J. Visser. Grammars as Contracts. In *Proceedings, Generative and Component-based Software Engineering (GCSE'00)*, volume 2177 of *LNCS*, pages 85–99, Erfurt, Germany, Oct. 2000. Springer.
51. B. Kadhim and W. Waite. Maptool—supporting modular syntax development. In T. Gyimothy, editor, *Proceedings, Compiler Construction (CC'96)*, volume 1060 of *LNCS*, pages 268–280. Springer, Apr. 1996.
52. G. Kiczales. The Fun has Just Begun. AOSD'03 Keynote Address, available from http://www.cs.ubc.ca/~gregor, 2003.
53. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming (ECOOP'901)*, pages 327–353, 2001.
54. P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380, 2005.
55. D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968. Corrections in 5:95-96, 1971.
56. J. Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proceedings, Source Code Analysis and Manipulation (SCAM'03)*, pages 161–172, Amsterdam, Sept. 2003. IEEE Computer Society Press.
57. J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In M. van den Brand and R. Lämmel, editors, *Proceedings, Language Descriptions, Tools, and Applications (LDTA'02)*, volume 65 of *ENTCS*. Elsevier Science, Apr. 2002. 7 pages.
58. R. Koschke and J.-F. Girard. An intermediate representation for reverse engineering analyses. In *Proceedings, Working Conference on Reverse Engineering (WCRE'98)*, pages 241–250. IEEE Computer Society Press, Oct. 1998.
59. G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug. 1988.
60. L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar. Consistency Problems in UML-based Software Development, 2002. Workshop proceedings; Research Report 2002:06.
61. R. Lämmel. Coupled Software Transformations (Extended Abstract). In *Proceedings of the First International Workshop on Software Evolution Transformations*, Nov. 2004. 5 pages; Online proceedings available at http://banff.cs.queensu.ca/set2004/.
62. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI'03.
63. R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 244–255, Snowbird, Utah, Sept. 2004. ACM Press.

64. R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M. van den Brand and D. Parigot, editors, *Proceedings, Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *ENTCS*. Elsevier Science, Apr. 2001.
65. B. S. Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.
66. B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM Press.
67. C.-T. Liu, P. K. Chrysanthis, and S.-K. Chang. Database Schema Evolution through the Specification and Maintenance of Changes on Entities and Relationships. In P. Loucopoulos, editor, *Entity-Relationship Approach - ER'94, Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach, Manchester, U.K., December 13-16, 1994, Proceedings*, volume 881 of *LNCS*, pages 132–151. Springer, 1994.
68. W. Lohmann, G. Riedewald, and M. Stoy. Semantics-preserving migration of semantic rules after left recursion removal in attribute grammars. In *Proceedings of 4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004)*, volume 110 of *ENTCS*, pages 133–148. Elsevier Science, 2004.
69. E. Mamas and K. Kontogiannis. Towards portable source code representations using XML. In *Proceedings, Working Conference on Reverse Engineering (WCRE'00)*, pages 172–182. IEEE Computer Society Press, Nov. 2000.
70. B. McLaughlin. *Java and XML data binding*. Nutshell handbook. O'Reilly & Associates, Inc., 2002.
71. E. Meijer and W. Schulte. Unifying tables, objects and documents. In *Proceedings of Declarative Programming in the Context of OO Languages (DP-COOL)*, Sept. 2003.
72. E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. In *XML Conference and Exposition*, Dec. 2003.
73. Microsoft Corp. The LINQ Project, 2005. http://msdn.microsoft.com/netframework/future/linq/.
74. C. Morgan. *Programming from Specifications*. Prentice Hall International, 1990.
75. J. Noble. Basic relationship patterns. In *Second European Conference on Pattern Languages of Programming*, 1997. Siemens Technical Report.
76. J. Noble and J. Grundy. Explicit Relationships in Object Oriented Development. In C. Mingins, R. Duke, and B. Meyer, editors, *Proceedings of TOOLS 18: Technology of Object-Oriented Languages and Systems Conference*, pages 211–225. Prentice Hall, Sept. 1995.
77. 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.
78. J. Oliveira. Software Reification using the SETS Calculus. In *Proceedings of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer, 8–10 January 1992.
79. J. Oliveira. Calculate databases with 'simplicity', Sept. 2004. Presentation at the IFIP WG 2.1 #59 Meeting, Nottingham, UK.
80. W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
81. J. Paakki. Attribute Grammar Paradigms — A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.
82. J. Park and S. Ram. Information systems interoperability: What lies beneath? *ACM Transactions on Information Systems*, 22(4):595–632, 2004.
83. J. Purtilo and J. Callahan. Parse tree annotations. *Communications of the ACM*, 32(12):1467–1477, 1989.

84. K.-U. Sattler, I. Geist, and E. Schallehn. Concept-based querying in mediator systems. *The VLDB Journal*, 14(1):97–111, 2005.
85. B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, pages 19–25, Sept./Oct. 2003. Special Issue on Model-Driven Development.
86. J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 65–76, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
87. T. Sheard. Generic unification via two-level types and parameterized modules. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 86–97, New York, NY, USA, 2001. ACM Press.
88. C. Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., Sept. 1995. Available from `http://citeseer.nj.nec.com/simonyi95death.html`.
89. T. Spitta and F. Werner. Die Wiederverwendung von Daten in SAP R/3. *Information Management & Consulting (IM)*, 15:51–56, 2000. In German.
90. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.
91. Sun Microsystems. The Java architecture for XML binding (JAXB). `http://java.sun.com/xml/jaxb`, 2001.
92. D. Thomas. The Impedance Imperative Tuples + Objects + Infosets = Too Much Stuff! *Journal of Object Technology*, 2(5):7–12, Sept.–Oct. 2003. Online available at `http://www.jot.fm/jot/issues/issue_2003_09/column1/`.
93. M. Van De Vanter. Preserving the documentary structure of source code in language-based transformation tools. In *Proceedings, Source Code Analysis and Manipulation (SCAM'01)*. IEEE Computer Society Press, 2001.
94. W3C. Document Object Model (DOM), 1997–2003. `http://www.w3.org/DOM/`.
95. W3C. XML Information Set (Second Edition), 1999–2004. `http://www.w3.org/TR/xml-infoset/`.
96. W3C. Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation, Feb. 2004. `http://www.w3.org/TR/2004/REC-xml-20040204/`.
97. W3C. XML Schema: Component Designators, W3C Working Draft, 29 Mar. 2005. `http://www.w3.org/TR/xmlschema-ref/`.
98. P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, New York, NY, USA, 1987. ACM Press.
99. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.
100. P. Walmsley. *Definitive XML Schema*. Prentice Hall, 2001. 556 pages, 1st edition.
101. A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, Jan. 1997.

A Exercises and riddles

We list exercises on a scale of '*' to '***'. Excellent, generalized solutions to the exercises in the three-stars category have the potential to lead to a workshop or conference paper. We also annotate exercises by 'G' to admit that googling might help, and we use 'P' for an indication that advanced programming skills are to be leveraged.

A.1 Mappings in parsing and un-parsing

We start with some old-fashioned mapping problems that work fine as warm-up. Parsers and un-parsers, at some level of abstraction, describe highly systematic mappings. However, occasionally, these mappings need to work hard to bypass a kind of 'impedance mismatch' between concrete and abstract syntax representations, or they need to account for implementational restrictions.

Exercise 1 (, G).* Given is a set of binary operators with associated priorities. Using your programming language of choice, give a concise description of a mapping that parses a list of operators and operands into the correctly parenthesized term. (Note that the actual operators and their priorities are a parameter of the mapping.) For instance, the list `[1,+,2,*,3]` should be parsed into the term `'+'(1,'*'(2,3))` assuming common priorities for `'+'` and `'*'`.

Exercise 2 (, G).* Continue Ex. 1 to include explicitly parenthesized expressions.

*Exercise 3 (**, G?).* Continue Ex. 2 as follows. Given is a term. Describe an 'un-parsing' mapping that generates the concrete representation (a list of operators and operands) with the minimum number of necessary parentheses.

*Exercise 4 (**, G?).* Continue Ex. 3 so that it will definitely preserve all parentheses that were explicit in the original input. That is, the composition of parsing and un-parsing should be the identity function on the set of all parseable strings. (Hint: the term representation needs to be refined.)

*Exercise 5 (**,G,P).* Here is a definite clause grammar (DCG) for the language $(a|b)^*$

```
% Prolog/DCG code
aorbs(snoc(Xs,X)) --> aorbs(Xs), aorb(X).
aorbs(lin)        --> [].
aorb(a)           --> [a].
aorb(b)           --> [b].
```

The grammar also describes the synthesis of a *left-associative* list (cf. snoc rather than cons and lin rather than nil). Such left-associativity suggests a left-recursive grammar, as shown indeed. However, Prolog's normal left-to-right computation rule implies non-termination for left recursion. Hence, we need a right-recursive grammar. Assignment: develop the corresponding DCG.

We could build an intermediate cons list, and rephrase it eventually:

```
aorbs(SL) --> aorbsCons(CL), { rephrase(CL,SL) }.
aorbsCons(cons(X,Xs)) --> aorb(X), aorbsCons(Xs).
aorbsCons(nil)        --> [].

rephrase(CL,SL) :- rephrase(CL,lin,SL).
rephrase(nil,SL,SL).
rephrase(cons(X,Xs),SL1,SL2) :- rephrase(Xs,snoc(SL1,X),SL2).
```

This solution involves an unnecessary traversal. We ask for a solution that avoids such an inefficiency. As an aside, general descriptions of left-recursion removal are available [3,68]. Also, one may consider techniques for deforestation [99], which could even be useful to automatically derive an efficient solution from the inefficient encoding that we have shown.

A.2 Mappings for XML grammars

When programming language folks first looked at DTD [96], some might have said "This is just a verbose variation on EBNF [47]." — leaving implicit that there are a few issues that go beyond context-free grammars, e.g., IDREFs. This proposition does not so easily generalize to the XML schema language (XSD), which is a relatively rich XML grammar formalism. In general, the differences between grammar notations (XML schema, DTD, Relax NG, Schematron, EBNF, BNF, SDF, ASDL, ASN.1, ...) invite insightful mapping exercises. Some XSD-based riddles follow.

Exercise 6 (,G?).* The EBNF formalism is orthogonal in itself in the sense that it offers regular operators that can be applied to other grammar phrases in arbitrary ways. In what sense does XML schema deviate here? (Hint: think of occurrence constraints.) Argue regarding the pros (if any) and cons of this deviation.

Exercise 7 (**). The content model <choice/> (i.e., the empty choice) is invalid according to the XML Schema recommendation by the W3C. Why is that a sensible restriction, and how does the notion of empty choice transcribe to context-free grammars? Suppose <choice/> was not forbidden, how does it compare to <sequence/>, and again, what does this comparison mean in context-free grammar terms? Give a few more algebraic equations on content models. For instance, give equations that involve occurrence constraints.

Exercise 8 (**). Consider the following schema:

```
<!-- XML schema -->
<xs:schema ... elided for brevity ...>
  <xs:element name="foo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="bar" type="xs:string"/>
        <xs:element ref="foo"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

According to the WC3 recommendation, this schema is valid. When considered as a context-free grammar, what basic property is violated by this schema? (Hint: it is the same property that is violated by <choice/>.) Argue that this property is valuable from an XML-centric point of view. Also explain the formal means to enforce such a restriction by adopting context-free grammar techniques.

Exercise 9 (***). Provide a detailed mapping for, what you might call, 'DTDification' of XML schemas. (We choose this name to hint at the similar process of YACCification, where EBNF-like expressiveness is normalized to BNF-like notation [64].) That is, how can you compile away the extra expressiveness of XSD such that the resulting schemas can be mapped to DTDs rather directly. Argue that the resulting DTD accepts a 'reasonable' superset of the XML instances that are accepted by the initial schema.

A.3 Compensation of semantical impedance mismatches

Mapping operations on data models may lead us to semantical challenges (as opposed to merely typing mismatch challenges). The following exercises focus entirely on fundamental properties of language semantics in the context of data processing.

*Exercise 10 (**,G,P).* We will be concerned with the simulation of a lazy semantics. This is clearly necessary when we want to transcribe data-processing problems from a lazy encoding to a non-lazy encoding (i.e., perhaps to an eager language). Consider the following Haskell session that demonstrates lazy list processing:

```
haskell> take 10 [0..]
 [0,1,2,3,4,5,6,7,8,9]
```

For the record, the function take is defined in the Haskell Prelude as follows:

```
-- Haskell 98 code
take n _       | n <= 0  =  []
take _ []                =  []
take n (x:xs)            =  x : take (n-1) xs
```

Also, the notation [0..] is a shortcut for incForever 0, where:

```
incForever n = n : (incForever (n+1))
```

Detailed assignments:

- Redefine lists, take and incForever such that an eager semantics would be sufficient.
- Transcribe the eager Haskell solution to an OO language as verbatim as possible.
- Use streams (as of C# 2.0 etc.), i.e., lazy lists, instead.
- Describe a data-processing scenario that calls for lazy structures other than lists.

*Exercise 11 (***,P).* We want to do data processing in Haskell with an OO-like reference semantics. Consider the following algebraic data types, given in Haskell syntax; they describe a fragment of an abstract syntax for an imperative, statement-oriented language with nested scopes for declarations of variables and procedures:

```
type Block = ([Dec],[Stm])
data Dec   = VarDec Id Type | ...                 -- procedures etc.
data Id    = Id String
data Type  = IntType | StringType
data Stm   = Assign Id Exp | BlockStm Block | ...  -- other statements
data Exp   = Var Id | ...
```

Hence, blocks in this language are lists of statements combined with the new declarations for this block. Each block opens a nested lexical scope. Now let us assume that we are interested in a richer AST format, which faithfully models ref/dec relationships. That is, whenever a variable is referenced in an expression or a statement, we want to be able to navigate from such a 'ref' side to the corresponding 'dec' side, i.e., to the binding block that holds the visible declaration.

Detailed assignments:

1. Extend the algebraic data types, given above, to include constructor components for ref/dec relationships. Take into account that these relationships may not be represented in terms initially, as they might be computed separately. Employ lazy, pure functional programming (rather than explicit references of the IO or the ST monad) to navigate from ref to dec sides.
2. Refactor the data types to use Haskell's IORefs. Illustrate the use of 'smart' constructors so that user code is not blurred by the allocation of references and assignments to references. A useful literature reference: [87].
3. How can we avoid cycles of *generic* algorithms that walk over the Haskell graphs? For instance, an algorithm for showing a Haskell term must not run into a cycle when hitting on a ref/dec relationship? Describe a technique that does not require intimate knowledge of the problem-specific data types.

A.4 XML, object, relational mapping

These exercises illustrate cross-paradigm impedance mismatches as discussed in Sec. 4.

Exercise 12 (*). Suppose we store XML documents with IDs and IDREFs in a relational database. What extra measures are necessary in case we want to (i) store multiple documents in the database, or (ii) extract new XML views from the database that potentially involve multiple documents?

Exercise 13 (**). Consider the following XSD identity constraints:

```
<!-- XML schema -->
<xs:element name="order" type="OrderType">
  <xs:keyref name="prodNumKeyRef" refer="prodNumKey">
    <xs:selector xpath="items/*"/>
    <xs:field  xpath="@number"/>
  </xs:keyref>
  <xs:key name="prodNumKey">
    <xs:selector xpath=".//product"/>
    <xs:field  xpath="number"/>
  </xs:key>
</xs:element>
```

These constraints read as follows: *"Each child of* items *must have a* number *attribute whose value is unique within the* order*. All* product *descendants of* order *must have a* number *child whose value matches one of these unique product numbers."* [100]. Let us assume that the schema with those constraints is bound to objects. How can we enforce the identity constraints within the object model?

Exercise 14 (**). We recall the discussion of cascading deletes in Sec. 4.3. A deletion of a stock item was supposed to lead to the deletion of all relevant transaction items. We seek the object-oriented counterpart for this cascading delete. Here, we assume that stock items and transactions reside in OO collections whose implementation has to be made aware of cascading.

To hint at the solution, we provide an SQL-based encoding that does not use the cascading annotations that we facilitated in Sec. 4.3. Instead, we create a trigger to kick in when a delete operation is about to affect the stock table:

```
// SQL Server 2000 code
CREATE TRIGGER stock_cascade_delete ON stock FOR DELETE AS
 DELETE FROM stock_trans
 WHERE stock_id IN
               ( SELECT stock_id FROM deleted )
```

Provide an OO encoding of the cascading behavior.

Exercise 15 (***). Continue Ex. 14 as follows. We seek a general, aspect-oriented solution that can be reused for cascading deletion. To this end, we note that the overall problem is similar to design patterns like ‘observer’ for which indeed modular, AOP-based solutions have been proposed [38,41]. Such an AOP-based solution may illustrate whether AOP can be useful for mastering cross-paradigm impedance mismatches.