Compatibility of XML Language Versions*

Daniel Dui and Wolfgang Emmerich

Department of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
{D.Dui|W.Emmerich}@cs.ucl.ac.uk

Abstract. Individual organisations as well as industry consortia are currently defining application and domain-specific languages using the eXtended Markup Language (XML) standard of the World Wide Web Consortium (W3C). This trend introduces new challenges for version and configuration management. We show that configuration management for XML languages is considerably more complicated for an XML Schema or DTD than it is for traditional software engineering artifacts. In addition to internal consistency of the language definition, also consistency between the language and its instance XML documents needs to be preserved when evolving the language definition. We propose a definition for compatibility between versions of XML languages that takes this additional need into account. Compatibility between XML languages in general is undecidable. We argue that the problem can become tractable using heuristic methods if the two languages are related in a version history. We propose to evaluate the method by using different versions of the Financial products Markup Language (FpML) in whose definition we participate.

1 Introduction

The eXtensible Markup Language (XML) is a meta-language for defining markup languages. It became a recommendation of the World Wide Web Consortium (W3C) in February 1998 [5] and since then it has gained enormous popularity. An XML document is simply a text file containing tags that identify its semantical structure. The XML specification [6] precisely defines the lexical syntax, or in XML-parlance "well-formedness", of a document. The well-formedness constraints impose what characters are allowed in an document, that for all open tags there shall be a corresponding closing tag, etc.

Satisfaction of well-formedness constraints alone is sufficient for a document to be parsed and processed by a variety of libraries and tools, but for most non-trivial applications the language designer will want to define a grammar for the language explicitly and precisely. She can define a concrete syntax by means of a schema language and a context-sensitive syntax by means of a constraint language.

* This work is partially funded by UBS Warburg.

The most common schema languages are currently Document Type Definition (DTD), XML Schema [12], and Relax NG. The DTD language is part of the XML 1.0 specification, it is simple, but of limited expressiveness. XML Schema and Relax NG have gained acceptance more recently; they are more expressive than DTD and support, among other things, data types and inheritance. Unlike DTD, they are themselves XML-based languages. XML Schema is a W3C recommendation as of May 2001 and Relax NG is currently an ISO draft standard.

Static semantic constraints can be specified with languages like Schematron [14] or the xlinkit [20] rule language. Schematron is a rule-based validation language that allows to define assertions on tree-patterns in a document and it is undergoing ISO standardisation at the time of writing. The xlinkit rule language is part of xlinkit, a generic technology for managing the consistency of distributed documents, that was successfully used to specify context-sensitive constraints for complex financial documents [10]. There is not currently a generic constraint language endorsed by the W3C.

A large number of organisations are currently using XML to define data formats. The data format can be a simple file format used by one single application or it could be a complex interchange format standardised by many organisations that constantly produce, store, and exchange innumerable instance documents. XML-based languages have been developed to represent chemical structures, gene sequences, financial products, B2B transactions, and complex software engineering design documents. We consider the definition of an XML-based language consisting of a document schema definition, given by means of a schema language, and of a set of additional constraints, given by means of a constraint language.

Some of these XML languages tend to evolve over time, for example because the initial requirements for the language have not been fully understood, or because change is inherent in the domain. We actively participate in the definition of the Financial products Markup Language (FpML), a language used to represent derivative transactions. In this domain, new financial products are being invented constantly and as a result FpML is in a constant state of flux. These changes need to be exercised in a controlled way and give raise to the need for version and configuration management of XML languages.

The main contribution of this paper is the observation that the version and configuration management needs for XML languages are different and more demanding than those of more traditional software engineering artifacts. We define the notion of compatibility between XML language definitions and show that in general compatibility is undecidable. We propose a heuristic method that exploits relationships between different versions of a language definition to decide version compatibility. We propose to evaluate the method using different versions of FpML.

The paper is further structured as follows. In Section 2, we give a more detailed motivation for the problem. Section 3, we define the notion of version compatibility both for language grammars and for static semantic constraints and show why compatibility is undecidable in general. In Section 4, we sketch our

heuristic method to solve compatibility between versions of the same language. We review related work in Section 5. We discuss further work and conclude the paper in Section 6.

2 Motivation

Our work on version and configuration management of XML languages is motivated by our participation in the FpML standardisation effort. The FpML language is both large and complex because FpML is used to represent derivative transactions, some of the most complex types of transactions traded in financial markets.

FpML raises several interesting questions for version and configuration management because it changes quickly over time for the following reasons: Firstly, FpML is being developed in a truly distributed manner by a number of different working groups that work concurrently on the language and therefore need to manage different versions appropriately. Secondly, the standard committee is including in the language support for the various types of financial derivative products gradually, as the interest for FpML grows, rather than attempting to include support for all of them at once. Thirdly, financial organisations constantly invent new products that they will want to represent in FpML either by changing the standard or by adding in-house extensions. Finally, it is inevitable that, as the language evolves, some of its parts will be redesigned to allow further developments or to mend previous mistakes.

The designers of FpML, and of many other complex XML languages, may need to make changes to the language while retaining overall compatibility. Intuitively and informally, compatibility demands first changes to the language to obey the syntactic and static semantic rules of the meta languages (such as DTDs, Schemas or constraint languages) and second the continued ability to validate any instance documents against the language. This validation would include both syntactic validation (against the schema of the language) and static semantic validation (against the constraint language).

Thus, the notion of compatibility for XML languages is wider than the one with which software configuration management was traditionally concerned. Unlike in usual software configuration management where the notion of compatibility can be established by examining a well-known and finite set of artifacts (such as design documents, code, deployment descriptors and test data), testing compatibility between XML languages typically involves an unknown and potentially infinite set of instances of that language.

It may be impractical to demand compatibility of language changes at all times. If, however, designers must introduce changes that break language compatibility, they will want to do this deliberately rather than accidentally and they would also need to convert instance documents between versions of the language. If also this is not a viable option, they will need to identify exactly what causes the incompatibility, which instance documents are affected and in what

way. And they can do this only with the assistance of appropriate methods and tools, which currently do not exist.

3 Compatibility

The aim of this section is to define more formally the notion of XML language compatibility that is absent from the existing XML specifications [6, 12, 2, 3].

The language definition is given by a schema that defines the concrete syntax and a set of consistency rules that define the static semantics for the language. An instance document is *valid* against the language definition if it satisfies all the constraints defined by the language schema and by the consistency rules. We note that XML schema and existing constraint languages, such as Schematron or xlinkit are XML languages themselves. Thus any modification to the language definition first of all has to be valid against the meta-constraints.

We can obtain several definitions of compatibility by reasoning on the relationship between *extents* of two languages. We borrow the term *extent* from the literature on object oriented databases [1] where it denotes the set of instances of a class. In the context of XML-based languages we use it to denote the set of all possible instance documents valid against a language definition. In most cases that occur in practice, this is an infinite set.

3.1 Syntactic Compatibility

We start with syntactic compatibility that only considers the schema. The simplest case of syntactic compatibility is when, taken two languages, the first is fully compatible with the second one. This happens when all possible instance documents of the first language are valid also with respect to the second language. In other words the extent of the first language is a subset of the extent of the second language. Figure 1 shows a Venn diagram where the sets A and B represent the extent of the two languages respectively. More formally:

Definition 1. *Let $L(A)$ be the extent of Schema A and $L(B)$ be the extent of Schema B. Schema B is syntactically fully compatible with Schema A if and only if $L(A) \subseteq L(B)$.*

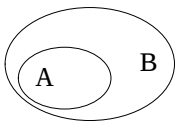

Fig. 1. Instance document sets for compatible languages

The compatibility relation is asymmetric: the fact that Schema B is compatible with Schema A does not imply that Schema A is compatible with Schema

B, which would happen only if $L(A) = L(B)$. We can also say that Schema B is backward compatible with Schema A when B is a new updated version of A.

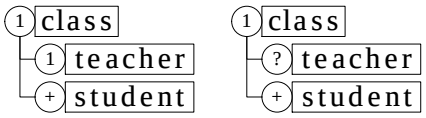

Fig. 2. Compatible schemas

Figure 2 gives an example of two Schemas A and B where B is compatible with A. This and the following examples in this section assume for simplicity that the schema fully defines the syntax of the language. Schema A defines that instance documents shall have exactly one element called `class` and inside that element there shall be exactly one element called `teacher` and one or more elements called `student`. Schema B is a new version version of Schema A. The only difference between the two schemas is the cardinality of element `teacher`, which in Schema B can appear zero or one time. Clearly all valid instance documents for Schema A will be valid also against Schema B, therefore Schema B is backward compatible with Schema A.

On the contrary, two schemas are syntactically incompatible if the subset relation between extents does not hold as Figure 3 shows. Formally this means:

Definition 2. *Schema B is incompatible with Schema A iff* $\exists_{b \in L(B)} b \notin L(A)$.

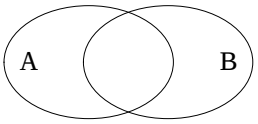

Fig. 3. Incompatible Languages

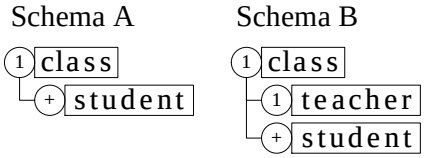

Fig. 4. Incompatible schemas

Figure 4 shows an example of two incompatible schemas. Schema B, as before, introduces inside the `class` element another element called `teacher`, but this time the new element must appear exactly once. All instance documents of Schema A do not have a `teacher` element, whereas all instance document of Schema B are required to have one. Schema B is therefore incompatible with Schema A.

Language incompatibility is usually an inconvenience, but it can be overcome if the designer can devise a transformation function that converts a instance documents for schema A to a valid instance document for Schema B.

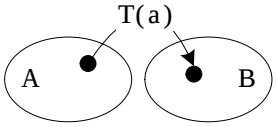

Fig. 5. Instance document transformation

Figure 5 shows how the transformation function $T(a)$ maps an element of $L(A)$ onto an element of $L(B)$.

```
Schema A instance document
<class>
  <student>Eric Cartman</student>
  <student>Kyle Broflovski</student>
  <student>Stan Marsh</student>
  ...
</class>

Schema B instance document
<class>
  <teacher>unknown<teacher>
  <student>Eric Cartman</student>
  <student>Kyle Broflovski</student>
  <student>Stan Marsh</student>
  ...
</class>
```

Fig. 6. Instance Document Transformation

Figure 6 gives an example of an instance document of Schema A to which a transformation is applied to convert it to an instance document of Schema B. The transformation defines to insert an element `teacher` with value "unknown" as a child of the element `class`.

We note that XML languages are context-free languages. This is because they require a push-down parser (rather than a finite state machine) to establish whether or not a document is valid against a schema. Equivalence, containment and empty intersection of context free languages have shown to be undecidable problems. For the proof of this undecidability, we refer to [22, 24] and note that therefore syntactic compatibility or incompatibility of general XML languages is undecidable.

Nevertheless, the examples in this section show that it can still be solved, at least in particular circumstances, which we will investigate in Section 4. In our case the languages in question are closely correlated because one is derived as a successor version from the other. We believe that this allows to find heuristic criteria to determine, in most practical scenarios, if two XML-based languages are compatible.

3.2 Static Semantic Compatibility

The previous subsection has dealt with the language syntax only as it is defined in the schema, but, in the general case, additional consistency rules are also part of the language definition. Examples of such constraints for the FpML language are given in [10]. Nonetheless, Definitions 1 and 2 still hold.

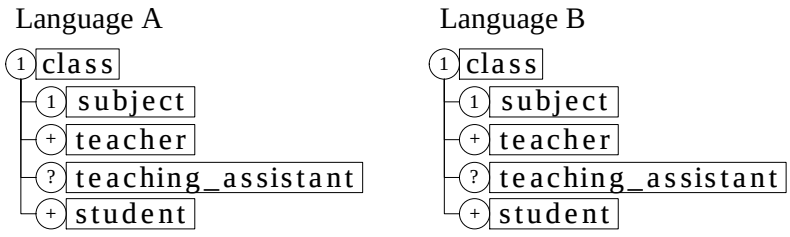

Fig. 7. Compatibility and consistency rules

Figure 7 shows an example of the evolution of a language definition that comprises a schema and a consistency rule. Rule B is *weaker* than Rule A so Language B is compatible with Language A.

In general, weakening constraints preserves compatibility, whereas strengthening constraints leads to incompatibility. The definition of the terms *stronger* and *weaker* in relation to constraints stems directly from logic:

Definition 3. *Let* $c1$ *and* $c2$ *be two constraints,* $c1$ *is termed* stronger *than* $c2$ *and* $c2$ weaker *than* $c1$ *iff* $c2 \Rightarrow c1$.

The static semantic constraints of a language are typically defined by a set of constraints. We can thus extend this definition to constraint sets in the following way:

Definition 4. *Let $\mathcal{A}$ be the set of constraints that determine for Language A and $\mathcal{B}$ be the set of constraints for Language B. $\mathcal{B}$ is static semantically compatible with $\mathcal{A}$ iff $\forall_{b \in \mathcal{B}} \exists_{a \in \mathcal{A}} b \Rightarrow a$.*

Another problem is that consistency rules for a language typically refer to the schema definition of the language, so there is also the problem of keeping them consistent with the schema as the schema evolves. For example if in Schema B the element `teaching_assistant` was renamed `assistant`, then also all rules that refer to that element would need to be updated accordingly. More precisely, the constraints of xlinkit or Schematron use XPath expressions. These expressions express traversals in the DOM tree, the syntax tree that XML parsers establish.

Definition 5. *We say that a static semantic constraint a is well-formed against a Schema A iff it only uses XPaths that are valid in A. We say that a set of constraints $\mathcal{A}$ is well-formed against a Schema A, iff $\forall_{a \in \mathcal{A}}$ a is well-formed against A.*

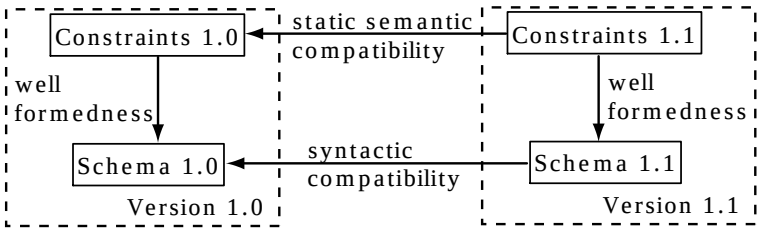

Fig. 8. Language Compatibility

We can now summarise our discussion and reformulate the problem of checking compatibility between language versions more precisely by reviewing Figure 8. An XML language definition consists of a schema that defines the grammar and a set of constraints that are well-formed against the grammar. In order for a new version of the language to be compatible to the previous version the following three conditions must hold:

- The new version of the schema must be syntactically compatible to its predecessor version.
- The new version of the constraint set must be well-formed against the new version of the schema.
- The new version of the constraint set must be static semantically compatible to the predecessor version.

4 Deciding Compatibility between Language Versions

We have already discussed that checking language compatibility is, in the general case, an undecidable problem. We note, however, that the definitions of two versions of the same language are bound to be strongly correlated. Our hypothesis is that it is possible to devise heuristic rules that allow to establish in most practical cases if one language definition is compatible or nor with another.

Checking well-formedness is not strictly speaking a version management problem as well-formedness also needs to be decided for single versions of a language definition. We therefore do not pursue this question further, but note that a well-formedness check needs to be executed when deciding on version compatibility of a change to a language.

For the two remaining problems, we propose the following approach. We establish the differences between both the constraints and the schema. As both the schema and the constraints are written in XML languages, we note that specialist algorithms can be used, such as XMLTreeDiff, which is based on a tree differencing algorithm [15] and delivers more precise results than text-based differencing. We then analyse the sets of differences.

A difference between versions can either be an addition of an element to a schema or a set of constraints, a deletion of an element of a schema or a constraint or a change to a schema element or a constraint. We discuss these separately now.

4.1 Syntactic Compatibility

We start the discussion of syntactic compatibility by reviewing the effect of changes that add or delete elements, the more straightforward cases. The addition of a new element to a schema by itself does not break syntactic compatibility. It may lead to a violation if another schema element is modified to include the new element, but then this case is handled by analysing the change to that element. If a schema element is deleted, it is clear that syntactic compatibility will be broken as instances of these types can then no longer occur in documents that are instances of the new version.

In order to analyse changes to an element of a schema, we can use the finite set of XML schema constructs to classify the change into those that retain compatibility and those that break it. Let us use the example in Figure 2 for illustrative purposes.

```
Schema A

<xsd:element name="class">
  <xsd:complextype>
    <xsd:sequence>
      <xsd:element name="teacher" type="xsd:string"
                   minOccurs="1" maxOccurs="1" />
      <xsd:element name="student" type="xsd:string"
                   minOccurs="1" maxOccurs="unbounded" />
```

```
    </xsd:sequence>
  </xsd:complextype>
</xsd:element>
```

Schema B

```
<xsd:element name="class">
  <xsd:complextype>
    <xsd:sequence>
      <xsd:element name="teacher" type="xsd:string"
                   minOccurs="0" maxOccurs="1" />
      <xsd:element name="student" type="xsd:string"/>
                   minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complextype>
</xsd:element>
```

Schema A defines an element called `class` that contains two other elements: `teacher` and `student`. Element `teacher` must occur in instance documents exactly once. Element `student` must occur at least once and the maximum number of its occurrences is unbound. Schema B differs from Schema A only because element `teacher` has been changed to be optional. This particular type of change modifies the cardinality in a compatible way as we can argue that the extent of the language for B includes the extent of A.

In general, we can classify changes into those that retain or violate syntactic compatibility. Examples of changes that retain compatibility include enabling fewer minimal set of elements in a sequence, increasing the number of maximal elements in a sequence, adding non-required attributes to an element, adding, changing or removing default initialisations, changing the order of elements in a choice and so on. Examples of changes to an element that do break syntactic compatibility include addition of a new non-optional sub-element to a sequence, changing the order of elements in a sequence, changing the type of an element or attribute, deleting an element from a sequence and so on.

In order to express these changes that do not break compatibility, we can now formalise these as constraints using the xlinkit constraint language at a meta level on XML schemas as follows:

```
<forall var="a" in="/Schema_A//element">
  <forall var="b" in="/Schema_B//element">
    <implies>
      <equals op1="$a/@name" op2="$b/@name" />
      <and>
        <equals op1="$a/@type" op2="$b/@type" />
        <and>
          <lessThanOrEq op1="$b/@minOccurs" op2="$a/@minOccurs" />
          <greaterThanOrEq op1="$b/@maxOccurs" op2="$a/@maxOccurs" />
        </and>
      </and>
```

```
    </implies>
  </forall>
</forall>
```

This rule imposes that for all elements of Schema A and Schema B that have the same name, the value of attribute `minOccurs` in Schema B is less than or equal to the value of the corresponding attribute `minOccurs` in Schema A and that the value of the attribute `maxOccurs` is greater than or equal to the value of the corresponding attribute `maxOccurs` in Schema A.

We can formalise rules for changes that would break syntactic compatibility in the same way. We can then use the xlinkit rule engine [20] to decide whether or not a particular version is syntactically compatible to a predecessor version.

4.2 Static Semantic Compatibility

Again we first review addition and deletion of constraints, which are the simple cases. Unless it is a tautology, the addition of a new constraint breaks static semantic compatibility because the new set of constraints will be more restrictive than the ones that were demanded in the predecessor version. The removal of a constraint will preserve static semantic compatibility as any document that was valid against a more inclusive set of constraints will continue to be valid against the smaller set of constraints.

It is possible to change a constraint without changing its meaning at all by using de Morgan's laws. By encoding these laws in a term rewriting system, we can check whether a particular change leads to an equivalent rule. For those rules where this is not the case, we need to establish whether they are weaker or stronger.

To analyse the implication of a change in that respect we can again use a similar approach that uses the constructs of the constraint language to classify whether or not a change weakens or strengthens a constraint. As an example, consider the following two xlinkit constraints, which in fact formalise the constraints shown in Figure 7.

```
Rule A

<forall var="a" in="//class">
  <implies>
    <or>
      <equals op1="$a/subject/text()" op2="'English'" />
      <equals op1="$a/subject/text()" op2="'French'" />
    </or>
    <exists var="b" in="$a/teaching_assistant" />
  </implies>
</forall>

Rule B
```

```
<forall var="a" in="//class">
  <implies>
    <equals op1="$a/subject/text()" op2="'English'" />
    <exists var="b" in="$a/teaching_assistant" />
  </implies>
</forall>
```

The xlinkit constraint language is a rather simple language, that can express boolean operators (and, or, implication and not), a few set of operators on DOM nodes (such as equal), as well as existential and universal quantification over a set of nodes identified by XPaths. Thus again we can identify for each possible change to an expression in a constraint whether it strengthens or weakens the constraint and then decide whether the overall change to the constraint breaks static semantic compatibility.

In the above example the change has strengthened the pre-condition of the implication by removing an `<or>` operand, thus making it less likely for the pre-condition to be true. Due to the *ex falso quod libet* rule for implications, this means that it is more likely for the overall formula to be true, which means that this change from version A to version B has weakened the constraint and is therefore statically semantically compatible.

Other examples of weakening a constraint include removing an `<and>` operand, adding an `<or>` operand, changing a universal quantifier into an existential quantifier, and so on. Examples of strengthening a constraint include adding an `<and>` operand, removing an `<or>` operand changing existential into universal quantification and so on.

5 Related Work

The problem of versioning and of managing a configuration of modules falls into the realm of software configuration management (SCM). According to Jacky Estublier [11], most issues about versioning have already been solved in the general case and the key problem is to incorporate these solutions into SCM tools. Reidar Conradi and Bernhard Westfechtel [8] give a comprehensive overview and classification of versioning paradigms. In their words: "In SCM systems, versioning of the schema is rarely considered seriously. On the other hand schema versioning often does not take versioning of instance data into account". Our approach intends to address both these concerns.

Most XML technologies take a programmer's pragmatic viewpoint and in some respects lack of formal foundation or consistency among them. We have already mentioned that the concept of schema compatibility is absent.

There is also some important work on formal analysis of XML schema languages [7, 16, 19, 17] part of which has been incorporated in XML standards such as Relax NG. However, none of these covers compatibility between language versions.

XML Schemas bear clear similarities with database schemas and collections of instance documents can be regarded as a data repository. Understanding of

database technology should provide insight on how to approach problems in the XML domain.

The most popular types of database systems are currently relational, object relational, and to a less extent object-oriented for which both the theory and the technology are mature and well documented in many books [23, 18, 9]. More recently, XML databases have appeared, where XML support is build into the data base management system (DBMS) [4].

We are interested in particular in schema evolution and schema versioning. Ferrandina et al. [13] have proposed a mechanism for schema evolution based on transition functions in the context of object-oriented databases. The difference to our work is that the extent of a class in an object-oriented schema is known – they can work on a closed world assumption, whereas for an XML language the extent is generally unknown and unaccessible.

6 Conclusions and Further Work

This paper identifies a novel area of research in software configuration management that will become increasingly important as the adoption of XML progresses. We have defined the notion of compatibility between different versions of an XML language for both the syntactic and the static semantic constraints of a language. We have observed that the problem of syntactic compatibility is undecidable in general, but have argued that it can become tractable by taking information about differences between versions into account.

Our future work will focus on refining and evaluating the approach for checking version compatibility that we were only able to sketch in this paper. We are in the process of completing the definition of syntactic compatibility rules in the xlinkit constraint language and can then use the existing xlinkit engine for exercising syntactic version compatibility checks. We will have to implement a term rewriting system to apply the de Morgan rules to xlinkit to test for equivalence and implement the strengthening and weakening semantics. To do so, we will again be able to re-use a large portion of the xlinkit rule engine, which to date implements two different semantics for the language in order to generate xlinks and to generate automated repair actions [21].

We have already identified our evaluation case study, which will be to check different versions of the FpML language for compatibility. The International Swaps and Derivatives Association (ISDA) has so far defined three major versions of FpML and for each of these versions a number of minor versions exist that have been in transitional use prior to adoption of the major version. We are actively participating in the standardisation of FpML and have participated in the establishment of a validation working group that will define the static semantic constraints for FpML. Constraints for version 1.0 have already been specified using xlinkit [25] and a working group is currently adding constraints for Versions 2.0 and upward.

Acknowledgements

We are indebted to Matt Meinel, Bryan Thal, Steven Lord and Tom Carroll of UBS Warburg and the members of the FpML Architecture Working Group for drawing our attention to the significant version and configuration managements challenges of FpML in particular and XML languages in general. We would also like to thank Anthony Finkelstein and Chris Clack for their helpful comments on an earlier draft of this paper.

References

1. F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: the Story of O_2*. Morgan Kaufmann, 1992.
2. P. V. Biron and A. Malhotra. XML Schema Part 1: Structures. Recommendation http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, World Wide Web Consortium, MAY 2001.
3. P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. Recommendation http://www.w3.org/TR/xmlschema-2/REC-xmlschema-2-20010502/, World Wide Web Consortium, MAY 2001.
4. R. Bourret. Xml and databases. www.rpbourret.com/xml/XMLAndDatabases.htm.
5. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.
6. T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Recommendation http://www.w3.org/TR/2000/REC-xml-20001006, World Wide Web Consortium (W3C), October 2000.
7. A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL - a model for W3C XML schema. In *World Wide Web*, pages 191–200, 2001.
8. R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.
9. C.J. Date and H. Darwen. *Foundation for Object/Relational Databases: The Third Manifesto*. Addison-Wesley, 1998.
10. D. Dui, W. Emmerich, C. Nentwich, and B. Thal. Consistency Checking of Financial Derivatives Transactions. In *Objects, Components, Architectures, Services and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2003. To appear.
11. J. Estublier. Software configuration management: a roadmap. In *ICSE - Future of SE Track*, pages 279–289, 2000.
12. D. C. Fallside. XML Schema Part 0: Primer. Recommendation http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, World Wide Web Consortium, MAY 2001.
13. F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proc. of the 20^{th} Int. Conference on Very Large Databases, Santiago, Chile*, pages 261–272, 1994.
14. R. Jelliffe. The Schematron. http://www.ascc.net/xml/resource/schematron, 1998.
15. K.Tai. The Tree-to-Tree Correction Problem. *Journal of the ACM*, 29(3):422–433, 1979.

16. D. Lee and W. W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(3):76–87, 2000.
17. D. Lee, M. Mani, and M. Murata. *"Reasoning about XML Schema Languages using Formal Language Theory"*, 2000.
18. M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999.
19. M. Murata, D. Lee, and M. Mani. "Taxonomy of XML Schema Languages using Formal Language Theory". In *Extreme Markup Languages*, Montreal, Canada, 2001.
20. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.
21. C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *Proc. of the* 25^{th} *Int. Conference on Software Engineering, Portland, Oregon*. ACM Press, 2003. To appear.
22. V.J. Rayward-Smith. *A First Course in Computability*. Blackwell, 1986.
23. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2001.
24. M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1997.
25. B. Thal, W. Emmerich, S. Lord, D. Dui, and C. Nentwich. FpML Validation: Joint proposal from UBS Warburg, University College London, and Systemwire. http://www.fpml.org, June 25, 2002.