Developing and Analyzing XSDs through BonXai *

Wim Martens
Institut für Informatik
Universität Bayreuth

Frank Neven
Hasselt University and
Transnational University of
Limburg

Matthias Niewerth
Technical University of
Dortmund

Thomas Schwentick
Technical University of
Dortmund

ABSTRACT

BonXai is a versatile schema specification language expressively equivalent to XML Schema. It is not intended as a replacement for XML Schema but it can serve as an additional, user-friendly front-end. It offers a simple way and a lightweight syntax to specify the context of elements based on regular expressions rather than on types. In this demo we show the front-end capabilities of BonXai and exemplify its potential to offer a novel way to view existing XML Schema Definitions. In particular, we present several usage scenarios specifically targeted to showcase the ease of specifying, modifying, and understanding XML Schema Definitions through BonXai.

1. INTRODUCTION

Through its endorsement by the W3C, XML Schema [14] is nowadays adopted as the industry wide standard for the specification of XML schema languages. XML Schema can be considered as the replacement of DTDs with added expressivity and flexibility. Unfortunately, the latter has also a negative impact on usability. Indeed, while DTDs are praised for their simplicity, XML Schema is notoriously difficult. It is designed to be machine-readable rather than human-readable and its specification alone (i.e., Part 1) already consists of 100 pages of intricate text. Studies reveal that XML Schemas (henceforth, *XSDs*) used in practice hardly take advantage of the additional structural expressivity over DTDs [2]. Two possible explanations come to mind. First, it could be that users have only a limited understanding of the possibilities of XML Schema; and, secondly, maybe not much additional expressiveness is needed. BonXai is an attempt to answer both concerns, as is explained next.

BonXai is an XML schema specification language which possesses all features of XSDs, including its expressivity, while retaining the simplicity of DTDs [9]. To this end, BonXai allows users to express contexts for elements by simple patterns without the need to explicitly specify complex types.[1] The objective of BonXai is by no means to replace XML Schema, but rather to provide a simple adds as much additional complication beyond DTDs as needed. Therefore, BonXai can also be seen as a practical front-end for XML Schema, i.e., "XML Schema for human beings". Indeed, the automatic translation into (and from) XML Schema is an important feature which distinguishes BonXai from other schema languages for XML. While several good alternatives for XML Schema exist, most notably DSD, Schematron and Relax NG [5, 13, 4], each with their own user base, they cannot be directly compiled into XML Schema for the simple reason that they can define schemas that are not representable as XSDs where never intended to be, a front-end for XML Schema. An additional strength of BonXai is its solid foundation which is rooted in pattern-based DTDs [10, 11] and which facilitates reasoning and transformation algorithms [6, 8].

In this demo, we focus on the capabilities of BonXai as a front-end for XML Schema. Specifically, we exhibit BonXai as a vehicle to facilitate the specification, modification, and analysis of XML Schema Definitions. Especially its use as an analysis tool distinguishes BonXai from other schema languages and constitutes the main novelty of the system.

A detailed overview of the demo is given in Section 4. In Section 2, we discuss the specification language BonXai itself. In Section 3, we discuss the different system components underlying BonXai.

2. BONXAI BY ONE EXAMPLE

Due to page limit restrictions, we illustrate BonXai by means of only one (toy) example, which is adapted from [12]. Figure 1 represents a BonXai-schema in *compact syntax*[2] defining a shop selling new as well as used cars. Figure 2 displays a matching XML-document. Like a DTD, a BonXai schema is a collection of rules. The right-hand side of a rule denotes a content model as usual. A left-hand side is not merely a label, but can be a linear XPath expression or even

*We acknowledge the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599

[1] Although, when desired, types can still be used.

[2] A BonXai XML syntax and a DTD-like syntax are available as well.

```
default namespace http://myshop.com/namespace
grammar {
  roots { shop }
    shop        = { element used, element new }
    used        = { element cars }
    new         = { element cars }
    cars        = { (element car)* }
    //used//car = { attribute make {string},
                    element year, element mileage }
    //new//car  = { attribute make {string},
                                  element warranty }
    year        = { type integer }
    warranty    = { type integer }
    mileage     = { type integer }
}
```

Figure 1: A BonXai-example in compact syntax.

an arbitrary regular expression. The semantics is that for an XML-document to match the schema, the children of nodes in the document selected by a left-hand side expression when evaluated from the root, should match the content model denoted in the right-hand side of the rule. For instance, the rule

```
//used//car = { element year, element mileage }
```

stipulates that `car`-elements occurring below a `used`-element should have a left child labeled `year` and a right child labeled `mileage`, while the rule

```
//new//car  = { element warranty }
```

says that `car`-elements occurring below a `new`-element should have a `warranty` as a child element. As a shorthand, we write `shop` rather than `//shop` to select arbitrary `shop`-elements.

Clearly, DTDs cannot model the schema in Figure 1 as they cannot distinguish between different kinds of elements with the same element name (i.e., they can only define one content model for `car`, thereby allowing a `mileage` child for new cars). However, the BonXai schema is syntactically only slightly more complicated than a DTD. The distinction between used cars and new cars can simply be specified by the contexts of car elements: by properties of the label sequence from the root node to the element. Indeed, if the `car` element occurs below a `new` element, the car is new, if it occurs below a `used` element it is used. In [11], it was shown that the additional expressivity of XML Schema can *always* be obtained in this way, that is, by specifying (regular) conditions on the path from the root to nodes. The strength of BonXai is that it makes these regular contexts explicit through concise patterns rather than obscuring them through the use of types. In particular, it frees users from the burden to specify explicit types. However, types are not completely abandoned and it is *possible* to specify them by simply attaching them to a rule. In addition to the identification of regular contexts, BonXai also contains features directly inherited from XML Schema like simple types, groups, namespaces, constraints (key/keyref/unique), and schema imports. More details on BonXai can be found in [9].

3. SYSTEM COMPONENTS

```
<shop>
  <used>
    <cars>
      <car make="audi">
        <year>2006</year>
        <mileage>12000</mileage>
      </car>
      <car make="bmw">
        <year>2008</year>
        <mileage>8000</mileage>
      </car>
    </cars>
  </used>
  <new>
    <cars>
      <car make="mercedes">
        <warranty>3</warranty>
      </car>
      <car make="porsche">
        <warranty>4</warranty>
      </car>
      <car make="opel">
        <warranty>5</warranty>
      </car>
    </cars>
  </new>
</shop>
```

Figure 2: An XML-document adhering to the schema in Figure 1.

Figure 3 presents a general overview of the system. The system is roughly divided into three parts, the Graphical User Interface(GUI), the actual BonXai library, and a general purpose automaton library.

3.1 User Interface

The GUI is provided through a plugin for the open source editor JEdit [7]. JEdit provides basic text editing functionalities, syntax highlighting and a flexible plugin interface. The BonXai-Plugin provides the connection between the BonXai library and JEdit. There is also a command line client, which provides an easy way for batch conversion of schemas.

3.2 Automaton Library

The automaton library provides a solid automata-theoretic core which, in addition, allows for an easy integration of existing automaton based algorithms, like for instance XSD inference algorithms [3] or algorithms for repairing the unique particle attribution constraint of XML Schema [1].

3.3 BonXai Library

The BonXai library constitutes the heart of the system. It provides modules for the representation, import and export, and the conversion between DTD, XSD, and BonXai. It also supports the conversion of schemas into type-automata and the validation of XML against type-automata.[3]

[3]A type-automaton representation for XSDs was introduced in [10].

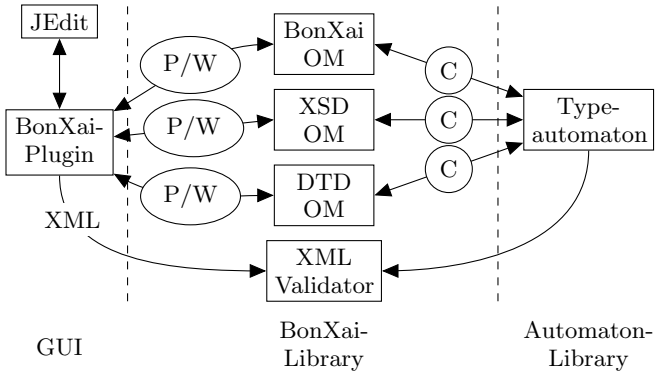

Figure 3: Schematic overview of the system components. P/W: Parser/Writer, C: Converter, OM: Object Model.

Import and Export

The modular structure of the library makes it easy to write new import and export modules. The system employs import and export modules writing java strings. The actual loading and storing of files is done by JEdit.

Object Models

Schemas are represented in an abstract way as type-automata. To facilitate manipulation of schemas, each class of schemas has its own object model. These object models store additional information, such as key, foreign key and uniqueness constraints, identifiers used for namespaces, typenames, etc.

XML Validator

The XML validator validates XML documents against type-automata and can thus be used to validate XML documents against BonXai schemas, DTDs, as well as XSDs.

Conversion Routines

As all conversions pass through the type-automaton representation, there are six conversion routines. The translation to and from XSDs and DTDs is rather direct. Computing a type-automaton from a BonXai schema, basically reduces to the construction of a product automaton encompassing all regular contexts in the schema. The converse direction requires to compute regular contexts for every state of the type-automaton. In addition, the conversion routine creates mappings between automaton states and the corresponding BonXai rules or XML schema types. This information together with the mappings between XML nodes and automaton states produced by the XML validator, is used by the GUI to highlight matching nodes/rules in the editor. Information about constraints and namespace identifiers is directly converted between the object models.

4. DEMO OVERVIEW

We will showcase the ability of BonXai to serve as a front-end for XML Schema through several use cases. In particular, we will exemplify how BonXai can be used to specify, modify, and analyze XSDs. Whereas the goal of the first use case is merely to introduce the language features of BonXai in support of XSD development, the goal of the remaining use cases is to show how BonXai opens up new ways to view and interpret XSDs by showcasing the use of BonXai as a tool for debugging, schema evolution, and analysis.

4.1 Specifying XSDs

As mentioned above, the purpose of this scenario is to familiarize users with the BonXai language. We will illustrate how BonXai can be used to specify XSDs from scratch. Thereto, we will create a BonXai schema in JEdit and translate it into an equivalent XSD keeping the features of the schema language in mind. In particular, this scenario will address the following issues:

(1) We show how BonXai seamlessly incorporates most of XML Schema language features (like differentiation between elements/attributes, simple types, element- and attribute groups, namespaces, constraints, schema imports, mixed types, default values, anytype/anyattribute) and demonstrate the three syntaxes for BonXai: compact syntax, XML syntax, and DTD syntax. Furthermore, we discuss how priorities for bonxai rules work to resolve conflicts. It is possible to define BonXai rules such that two or more rules match the same element. For example, both rules

```
//section = { element title, element paragraph+,
                              element section* }
//section/section/section = { element title,
                             element paragraph+ }
```

are matched by a `section` element that is below two other `section` elements. When such a multiple match occurs, BonXai gives priority to the rule that occurs lowest in the schema. So, in this case, the rule for

```
//section/section/section
```

would take priority. The rationale is that a developer could first write down rules that generally apply in the schema and write down the special cases and exceptions later. So, the two rules in the above example could be read as: *All sections have a title, a non-empty sequence of paragraphs, and a sequence of sections. But, we disallow nesting depths of sections larger than 3.* Two alternative ways for dealing with multiple matches of rules have been studied in the literature: *universal* and *existential* semantics [6, 8]. We do not adopt them here, since both options would damage the compatibility between BonXai and XML Schema.

(2) We demonstrate how the GUI aids to understand the correspondence between BonXai rules and the generated complex types in the transformed XSD. Although the simplicity of BonXai depends on the absence of complex types, *type names* can serve as short descriptions to help the user understand BonXai rules. In contrast to XML Schema, such type names have no semantic meaning whatsoever. Type names in BonXai are written in the (optional) comment blocks for rules. In the translation to XML Schema, type names can be helpful to generate names for XSD complex type definitions.

(3) Finally, we demonstrate advanced functionalities of our GUI to aid in schema development and -debugging. In particular, we support to inspect relationships between an XML document, a BonXai schema, and a corresponding XSD as follows:

- Highlighting of XML elements matched by a certain BonXai rule or by an XSD complex type.

- Highlighting of the rule/type matching an element in an XML tree.
- Highlighting the rule corresponding to an XSD complex type and vice versa.
- Finding nodes in an XML tree violating the schema.
- Finding nodes in an XML tree which are unconstrained by the schema.
- Finding unconstrained parts in a BonXai schema.

4.2 Modifying and analyzing XSDs

Here, we will show that BonXai is much more than only a nice syntax for XSDs. We exemplify BonXai as a tool for debugging, schema evolution, and analysis of XSDs. In particular, we showcase the following use cases:

Injection of BonXai into an existing XSD. We exemplify how an existing XSD can be altered by adding additional constraints specified in BonXai. The part of the XSD which is to be ported to BonXai is specified by highlighting the associated complex type that uniquely identifies the sub-schema of interest. We then add additional BonXai rules that describe the changes we want to perform in our schema and we inject the XSD translation of BonXai back into the original XSD at the right place. The highlighting features of the system, mapping patterns in BonXai rules to complex types in the generated XSD fragment, then provide the developer with control to inspect the induced changes in the original XSD more rapidly and accurately.

Upgrading DTDs to XSDs using BonXai. We illustrate how to transition from a simple DTD to a more refined and precise XSD to describe the targeted set of XML document. Taking advantage of the BonXai DTD-like syntax (which is conservative w.r.t. plain DTDs), we show how XML Schema features (like, e.g., simple types) and additional constraints that go beyond the expressiveness of the DTD language, can be added. The automatic translation to XSD and the relatively compact DTD-like syntax make this transition quite effortless.

Debugging invalid XML documents w.r.t. an XSD. When an XML document is invalid with respect to an XSD, BonXai can offer a transparent explanation when the mismatch is caused by a complex type violation. We will show the functionality of the system to convert the XSD to BonXai and highlight where an element mismatch occurs. The left-hand sides of the BonXai rules can offer more insight in terms of simple patterns for which kinds of elements are affected than the complex-type names provided by the XSD. Again, the highlighting features of the system can aid the developer to understand how changes in patterns affect the invalid XML document.

Analyzing real-world schemas. We will use a real-world schema and analyze it using a *BonXai-inspector*. We can load a specific part of an XSD into BonXai in order to analyze its structural complexity. Such a BonXai inspection could, e.g., give an idea of the amount of structural expressiveness which goes beyond DTDs and where it sits. In addition, the selection patterns provided by BonXai provide direct insight into the definition of elements depending on their context. As such, the BonXai translation, converting the machine readable syntax of XSDs in the more human-readable compact syntax of BonXai, and the associated highlighting features in our GUI help users to understand schema definitions more quickly and easily. Furthermore, we will show an additional feature that annotates complex types in XSDs with its corresponding BonXai selection pattern. This annotation gives users immediate insight on where a given complex type is used in an XML document. Since such selection patterns are basically specified in a fragment of XPath, users familiar with XML technology can already benefit from this feature without having to learn yet another standard.

5. REFERENCES

[1] Geert Jan Bex, Wouter Gelade, Wim Martens, and Frank Neven. Simplifying XML Schema: effortless handling of nondeterministic regular expressions. In *ACM International Conference on Management of Data (SIGMOD)*, pages 731–744, 2009.

[2] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML Schema: A practical study. In *International Workshop on the Web and Databases (WebDB)*, pages 79–84, 2004.

[3] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, 2010.

[4] James Clark and Makoto Murata. Relax NG specification. http://www.relaxng.org/spec-20011203.html, December 2001.

[5] Document structure description (dsd). http://www.brics.dk/DSD/.

[6] Wouter Gelade and Frank Neven. Succinctness of pattern-based schema languages for XML. *Journal of Computer and System Sciences*, 77(3):505–519, 2011.

[7] jEdit programmer's text editor. www.jedit.org.

[8] Gjergji Kasneci and Thomas Schwentick. The complexity of reasoning about pattern-based XML schemas. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 155–164, 2007.

[9] W. Martens, V. Mattick, M. Niewerth, S. Agarwal, N. Douib, O. Garbe, D. Günther, D. Oliana, J. Kroniger, F. Lücke, T. Melikoglu, K. Nordmann, G. Özen, T. Schlitt, L. Schmidt, J. Westhoff, and D. Wolff. Design of the BonXai schema language. Available at http://ls1-www.cs.tu-dortmund.de/cms/bonxai/, Draft 2011.

[10] Wim Martens, Frank Neven, and Thomas Schwentick. Simple off the shelf abstractions of XML Schema. *SIGMOD Record*, 36(3):15–22, 2007.

[11] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML Schema. *ACM Trans. Database Syst.*, 31(3):770–813, 2006.

[12] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 35–46, 2000.

[13] Schematron. http://www.schematron.com/.

[14] C.M. Sperberg-McQueen and H. Thompson. XML Schema. http://www.w3.org/XML/Schema, 2005.