BonXai: Combining the simplicity of DTD with the expressiveness of XML Schema

Wim Martens
Universität Bayreuth

Frank Neven
Hasselt University and transnational University of Limburg

Matthias Niewerth
Universität Bayreuth

Thomas Schwentick
TU Dortmund University

ABSTRACT

While the migration from DTD to XML Schema was driven by a need for increased expressivity and flexibility, the latter was also significantly more complex to use and understand. Whereas DTDs are characterized by their simplicity, XML Schema Definitions (XSDs) are notoriously difficult. In this paper, we introduce the XML specification language BonXai which possesses most features of XSDs, including its expressivity, while retaining the simplicity of DTDs. In brief, the latter is achieved by sacrificing the explicit use of types in favor of simple patterns expressing contexts for elements. The goal of BonXai is by no means to replace XML Schema, but rather to provide a simpler DTD-like alternative to schema designers that do not need the explicit use of types. Therefore, BonXai can be seen as a practical front-end for XML Schema. A particular strong point of BonXai is its solid foundation rooted in a decade of theoretical work around pattern-based schemas. We present in detail the formal model for BonXai and discuss translation algorithms to and from XML Schema.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Data Description Languages (DDL)*

General Terms

Design, Languages, Algorithms

Keywords

XML; BonXai; Schema Language

1. INTRODUCTION

Through its endorsement by the W3C, XML Schema [29] 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 regarding namespaces, modularization, and datatypes. As an unfortunate side effect, the migration to XML Schema 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 the central document of its specification (Part 1 of the specification) already consists of 100 pages of intricate text [12]. In their book, Møller and Schwartzbach discuss the comprehensibility of XML Schema as follows [23] (p. 156):

> *XML Schema is generally too complicated and hard to use by non-experts. This is a problem since many non-experts need to be able to read schemas to write valid instance documents.*

The goal of BonXai is to address this point by reconciling the expressivity and many features of XML Schema with the simplicity of DTDs. Whereas BonXai builds upon many ideas from existing schema languages, its most important feature, distinguishing itself from other schema languages, is its ability to serve as a front-end for XML Schema. Not only can BonXai schemas be readily transformed to and from XML Schema, but a BonXai schema itself can also be used to inspect, analyze and provide a deeper understanding of the corresponding XML Schema Definitions (henceforth, XSDs).

The purpose of this paper is to present a schema language for XML, called BonXai, specifically tailored as a practical schema design language, not to replace XML Schema but in support of the development of XML Schemas.

One of the most significant changes in the migration from DTDs to XML Schema is the introduction of types. The latter addition not only allows for a development style closely resembling object-oriented design and thereby facilitating modularization (for instance, through derivation and substitution), but types also significantly increase the structural expressiveness of schemas by allowing element definitions to depend on the context in which they appear. Surprisingly, studies reveal that XSDs occurring in practice hardly take advantage of the additional structural expressivity over DTDs [3]. In fact, most real world XSDs are structurally equivalent to a DTD. While the precise cause of the latter restricted use is unclear (we are not aware of any studies that tried to explain this), plausible explanations are that users do not know how to wield the extra expressiveness of XML Schema or that it is too cumbersome to write sophisticated and precise schemas when weighed against their

PODS'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2757-2/15/05 . . . $15.00.
http://dx.doi.org/10.1145/2745754.2745774.

obvious benefits. Actually, Møller and Schwartzbach assert that the introduction of types is a major aspect complicating the design of XSDs (pg. 156) [23]:

> *One important factor of the complexity of the language is the type mechanism. Even without type derivations and substitution groups, this notion of types adds an extra layer of complexity: an element in the instance document has a* name*, some element declaration in the schema then assigns a* type *to this element name, and finally, some type definition then gives us the* constraints *that must be satisfied for the given element. In DTD, an element name instead directly identifies the associated constraints.*

In other words, the use of types to express structural constraints could be beyond the average user. The main idea underlying BonXai is to remove the need to use types to express structural constraints by adding those constraints as primitives to the language. That is, BonXai allows users to express contexts for elements by simple patterns without the need to explicitly specify and define complex types. Regardless of why one believes that many XSDs in practice are structurally equivalent to DTDs, BonXai should make schema development and XSD development easier.

We stress once more that the objective of BonXai is by no means to replace XML Schema, but rather to provide a simple way to specify and manipulate a large class of XML Schemas that only 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, as already mentioned above, 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 [9, 28, 27], 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. We give a comparison with contemporary schema languages in Section 3.3.

An additional strength of BonXai is its solid foundation which is rooted in pattern-based schemas [20, 21] and which facilitates reasoning and transformation algorithms [13, 16]. Martens et al. [21] have shown that the increase in structural expressiveness from DTDs to XSDs lies in the ability to specify element definitions relative to a certain context. Whereas DTDs are restricted to element definitions relative to the name of the element, XSDs can specify element definitions relative to the path of element names from the root leading to that element.

We present a formal model for the core of BonXai and give formal descriptions of algorithms that translate back and forth between XML Schema and BonXai. These algorithms illustrate why the two languages are expressively equivalent. We analyze the worst-case blow-ups in these translations and show why our algorithms are worst-case optimal. Furthermore, we discuss practically relevant fragments of XML- and BonXai Schemas in which the conversions are particularly efficient.

As a reality check, we implemented the BonXai system in a tool [19] that allows, among other things, to parse BonXai schemas, validate XML against them and highlights matching rules, and can translate back and forth between BonXai and XML Schema.

Outline. This paper combines a practical language's exposition with an explanation of the underlying theory. We hope that in this way a reader who is familiar with either the practical or theoretical side can easily get an understanding of the other side as well. Section 2 provides a light-weight introduction to BonXai through a comparison with XML Schema that avoids notions from theoretical computer science and only requires a basic understanding of DTDs and XML Schema. In Section 3, we discuss more features of BonXai and its implementation, and consider its relationship with other XML schema languages. Section 4 introduces the formal model for BonXai and discusses the translations into and from XML Schema. We conclude in Section 5.

2. BONXAI BY EXAMPLE

In this section, we compare the ability of DTDs and XSDs to specify element definitions relative to a context and discuss how the latter influenced the design of BonXai.

Document Type Definitions (DTDs) constitute the first schema language for XML and are most well-known for their simplicity. Basically, DTDs are a grammar-based formalism where element declarations are entirely context insensitive. That is, the *content model* for an element is solely dependent on the name of that element.

We will now discuss a toy markup language that we will use to discuss the main features of XML Schema and BonXai. We first describe the markup language and an example document informally and then we will define a DTD, XML Schema, and BonXai schema for it.

Example 2.1 (An example document). *Consider the XML tree in Figure 1 with content formatted in a fictional markup language. The* `document` *is divided into three parts:* `template`, `userstyles` *(which contains user-defined style definitions), and* `content`*. The* `content` *part contains the actual text of the document, with markup (bold, font changes, etc.). Inside* `content`*, the text is structured by* `section` *elements, which can be nested to form subsections, etc.*

The `template` *element should describe the default formatting of the text within* `content`*. One could think that* `template` *defines ACM SIG style, for example. Within* `template`*, the default formatting of sections is specified within the* `section` *child of* `template` *and the default formatting of subsections within the* `section` *grandchild. So, a difference between* `template` *and* `content` *is that, in* `template`*, there is at most one* `section` *element per nesting depth. For the sake of the example, the rationale is that the default formatting of all sections at the same level should be the same. Furthermore,* `template` *does not contain text since all the actual text is within* `content`*.*

The `userstyles` *element contains a list of* `style` *elements. Each such* `style` *element should be thought of as being either some user-defined style (e.g., a fancy font for bold mathematics). Each* `style` *element has a unique name, which can be referred to from within* `content`*. Our example uses only one user-defined style:* `userdefined1`*.*

We chose our example such that it has elements within `content` and within `template` that have the same element names but different semantics, notably, the `section` element. Similarly, `style` has a different semantics if it is used within

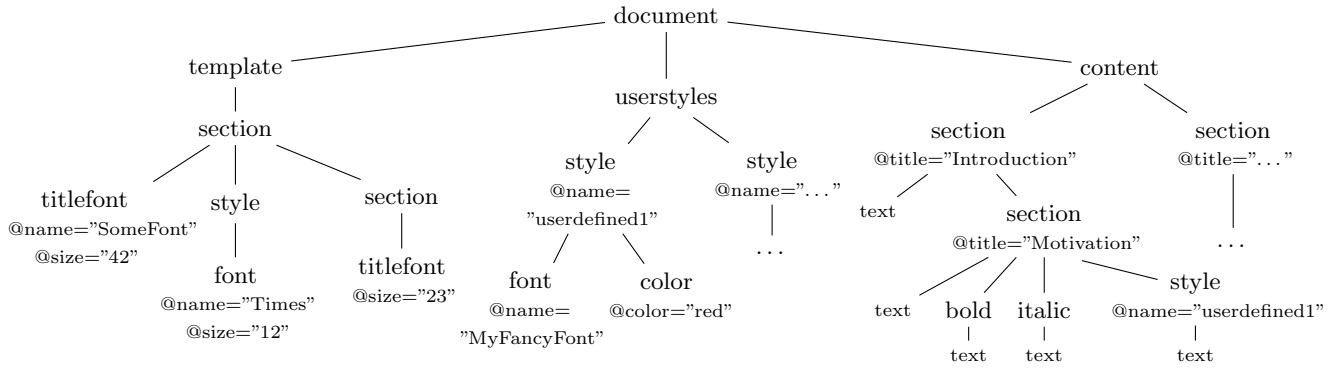

Figure 1: Example XML document.

```
<!ELEMENT document   (template, userstyles, content)>
<!ELEMENT template   section>
<!ELEMENT userstyles style*>
<!ELEMENT content    section*>
<!ENTITY % markup    "bold|italic|font|style|color">
<!ELEMENT section    (#PCDATA|titlefont|section|
                                  %markup;)*>
<!ATTLIST section    title CDATA #IMPLIED>
<!ELEMENT bold       (#PCDATA|%markup;)*>
<!ELEMENT italic     (#PCDATA|%markup;)*>
<!ELEMENT font       (#PCDATA|%markup;)*>
<!ATTLIST font       name CDATA #IMPLIED
                     size CDATA #IMPLIED>
<!ELEMENT style      (#PCDATA|%markup;)*>
<!ATTLIST style      name CDATA #IMPLIED>
<!ELEMENT titlefont  EMPTY>
<!ATTLIST titlefont  name CDATA #IMPLIED
                     size CDATA #IMPLIED>
<!ELEMENT color      (#PCDATA|%markup;)*>
<!ATTLIST color      color CDATA #REQUIRED>
```

Figure 2: A DTD describing the XML document in Figure 1.

`userstyles`, within `template`, or within `content`. DTDs do not have the expressive power to take these differences into account and must define a common content model for all elements with the same name. That is, a DTD can only define one rule for `section`, independent of where a `section` element occurs in the document.

Example 2.2 (DTD for Example 2.1). *A complete DTD for which the XML document is valid is given in Figure 2. Note the use of the entity* `markup` *that allows us to write the schema more succinctly. We present this entire DTD because it is instructive to compare it with the XSD which we expose next and with the BonXai schema which we define later and is equivalent to the XSD.*

We next develop an XSD for our example markup language which is able to differentiate the elements with the same name but different semantics. Specifically, XSDs can take context into account through the explicit use of types.

Example 2.3 (XSD for Example 2.1). *A fragment of an XSD for the markup language of Example 2.1 is presented in Figure 3. Figure 3 contains the definition of the root* `document` *node. Similar to the DTD, it has a group* `markup` *(at the end) to avoid any unnecessary verbosity. All our type names start with a capital* T *so that the reader can easily distinguish them from element names.*

The XSD distinguishes between two types *of sections:* Tsection *and* TtemplateSection*. The former should be used within* `content` *and the latter one within* `template`*. The* type *of a* `section` *element is determined by the type of its parent. That is, when the parent of such an element is labeled* `content` *or is a* `section` *element with type* Tsection*, the section can contain text and markup. On the other hand, if the parent is labeled* `template` *or is a* `section` *with type* TtemplateSection*, the* `section` *element cannot contain text, it can only contain formatting instructions. Similarly, the XSD should contain three types that can be used for* `style`*:* TtemplateStyle *(for* `style` *elements below* `template`*),* TnamedStyle *(for* `style` *elements below* `userstyles`*, and* TstyleRef *(for* `style` *elements below* `content`*).*

The tree representation of XML documents is crucial for understanding the expressiveness of XML Schema and, therefore, also the expressiveness of BonXai. Intuitively, XML Schema can distinguish between elements of the same name when they have different labels on the path to the root of the XML tree, the so-called *ancestor path*.[1] So, XML Schema can distinguish the `section` elements within `content` from those within `template`, for example. Indeed, the former have labels `section content document` on the path to the root, whereas the latter have `section template document`. (Similarly, XML Schema can also distinguish between `style` within `userstyles`, within `template`, or within `content`.) In [21], it was shown that the kind of constraint which can be put on such an ancestor path by an XSD can always be captured by a regular expression and, in over 98% of the XSDs in the study, even by so-called linear XPath expressions [16], which are Core XPath expressions that do not branch.[2] The latter insight influences the design of BonXai to make such contexts explicit through the addition of patterns over ancestor paths.

[1]This property of XML Schema originates from the Element Declarations Consistent constraint, which is enforced by the XML Schema Specification [12] (Section 3.8.6.3) prohibits the use of the same element occurring in the same content model with different types. A detailed discussion on the implications of this constraint can be found in [21, 20].

[2]Consequently, linear XPath expressions can only reason about *paths* in trees.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns="http://mydomain.org/namespace"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     elementFormDefault="qualified"
     targetNamespace="http://mydomain.org/namespace">
 <xs:element name="document">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="template">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="section" minOccurs="0"
                           type="TtemplateSection"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="userstyles">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="style" minOccurs="0"
           maxOccurs="unbounded" type="TnamedStyle"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="content">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="section" minOccurs="0"
              maxOccurs="unbounded" type="Tsection"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:complexType name="TtemplateSection">
  <xs:sequence>
   <xs:element name="titlefont" type="TtemplateFont"
                                       minOccurs="0"/>
   <xs:element name="style"    type="TtemplateStyle"
                                       minOccurs="0"/>
   <xs:element name="section"  type="TtemplateSection"
                                       minOccurs="0"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="Tsection" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
   <xs:group ref="markup"/>
   <xs:element name="section" type="Tsection"/>
  </xs:choice>
  <xs:attribute name="title" type="xs:string"
                                       use="required"/>
 </xs:complexType>
 <xs:group name="markup">
  [...]
 </xs:group>
 [...]
</xs:schema>
```

Figure 3: An XSD for the XML tree in Fig. 1.

We now discuss two BonXai schemas for the running example's markup language. The BonXai schema in Figure 4 is equivalent to the DTD given in Figure 2, while the BonXai schema in Figure 5 exploits the additional expressiveness of BonXai to be equivalent to the (full version of the) XSD of Figure 3.

Both examples use a compact syntax inspired by Relax NG [27]. Like a DTD, a BonXai schema is a collection of rules. The right-hand side of a rule denotes a content model as usual. The left-hand side can be either a label or a regular expression if more expressiveness is needed. We use a regular expression syntax which resembles XPath expressions since this allows users to also write linear XPath expression on left-hand sides. 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

```
template//section = { element titlefont?,
                      element style?, element section? }
```

stipulates that `section` elements occurring somewhere below a `template` element can contain a `titlefont` child, a `style` child, and a `section` child, whereas the rule

```
content//section = mixed {  attribute title,
                     (element section|group markup)*}
```

stipulates that elements occurring somewhere below a `content` element should contain a title and may contain text (indicated by the keyword `mixed`) with markup. The keyword `mixed` allows mixed content, i.e., it is allowed to interleave text with XML tags. In the BonXai schema in Figure 5, `/` and `//` stand for the XPath axes "child" and "descendant", respectively. We denote concatenation, disjunction, Kleene star, and "optional" by "`,`", "`|`", "`*`", and "`?`", as in DTDs. The operator "`&`" stands for unordered concatenation, which is known as `xs:all` in XSD. If an expression does not start with `/` or `//`, we implicitly assume that it starts with `//`. This way, simple lables are just a special case of regular expressions.

```
target namespace http://mydomain.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema
global { document }
groups {
 group markup = { element bold | element italic |
       element font | element style | element color }
}
grammar {
 document   = { element template, element userstyles,
                element content }
 template   = { element section }
 userstyles = { (element style)* }
 content    = { (element section)* }
 section = mixed { attribute title,(element section |
                  element titlefont | group markup)* }
 bold    = mixed { (group markup)* }
 italic  = mixed { (group markup)* }
 font    = mixed { attribute name, attribute size,
                   (group markup)* }
 style   = mixed { attribute name, (group markup)* }
 titlefont =     { attribute name, attribute size }
 color   = mixed { attribute color, (group markup)* }
 @name   = { type xs:string }
 @color  = { type xs:string }
 @title  = { type xs:string }
 @size   = { type xs:integer }
}
```

Figure 4: A BonXai schema equivalent to the DTD in Figure 2.

```
target namespace http://mydomain.org/namespace
namespace xs = http://www.w3.org/2001/XMLSchema
global { document }
groups {
 attribute-group fontattr = { attribute name?, attribute size? }
 group markup = { ( element bold | element italic | element font | element style | element color )* }
}
grammar {
 document                                = { element template, element userstyles, element content }
 content                                 = { (element section)* }
 template                                = { (element section)? }
 userstyles                              = { (element style)* }
 content//section                        = mixed { attribute title, (element section | group markup)* }
 content//style                          = mixed { attribute name, group markup }
 content//font                           = mixed { attribute-group fontattr, group markup }
 content//color                          = mixed { attribute color, group markup }
 (bold|italic)                           = mixed { group markup }
 template//section                       = { element titlefont?, element style?, element section? }
 template//style                         = { element font? & element color? }
 userstyles/style                        = { attribute name, element font? & element color? }
 (userstyles|template)//color            = { attribute color }
 (userstyles|template)//(font|titlefont) = { attribute-group fontattr }
 (@name| @color|@title)                  = { type xs:string }
 @size                                   = { type xs:integer }
}
```

Figure 5: A BonXai schema equivalent to the (partial) XSD from Figure 3.

The main difference with the corresponding XSD is that contexts are now defined explicitly. Another way of viewing the difference between XSD and BonXai is top-down versus bottom-up. XSDs carry all relevant information about the root-path in a top-down fashion, encoded in types, while BonXai, instead, looks upward from a node, thus separating types from their inference. Furthermore, as XSDs employ types, context has to be specified in terms of automata, while BonXai can use the more user-friendly regular expressions or linear XPath expressions.

3. BONXAI, THE PRACTICAL LANGUAGE

In Subsection 3.1, we present BonXai in more detail but do not intend to discuss every feature of the language and its relationship with XML Schema here. Instead, we provide a high-level overview and refer the reader to [18] for further details. We just discuss a few BonXai-specific matters (ancestor patterns, child patterns, and priorities) and then argue 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).

Subsection 3.2 explains BonXai's priority system. As mentioned in the introduction, the design of BonXai is influenced by existing XML schema languages. We discuss these in Subsection 3.3.

3.1 The BonXai Schema Specification Language

BonXai schemas consist of up to five blocks. First comes the *namespace block*, declaring all namespaces used in the schema. The second block is called the *global block* and specifies which element names can occur at the root of documents that match the schema. Third, there is an optional *group block*, which can declare the equivalent of XSD groups. The fourth block is called the *grammar block* and is the actual core of the schema. The grammar block contains the definitions of the rules that define the structure of documents. Finally, there is an optional *constraints block* which defines integrity constraints.

Global Element Names: Elements that are declared `global` in a BonXai schema can occur as root elements in XML documents that match the schema. In our running example, there is a single such element, called `document`.

Ancestor Patterns: A rule within the grammar-block of a BonXai schema is of the form

```
<ancestor pattern> = <child pattern>
```

The ancestor pattern (left of the equality sign) describes the context of the rule and should be matched against paths in the tree that start from the root. Ancestor patterns are variants of regular expressions, built from element names and attribute names (i.e. names starting with @). The regular expressions have the operators union (|), concatenation (/), descendant (//), Kleene star (∗), one-or-more (+), and zero-or-one (?). Sub-patterns can be grouped using round brackets. For compatibility with XML Schema, we need that, if attribute names appear, they occur at the end of ancestor patterns. For example, `(/a/a)*(@c|@d)` is allowed (and specifies `c`- and `d`-attributes for even-depth nodes that are labeled `a` and only have `a`-labeled ancestors), but `/a/@b/c` is not allowed. (Indeed, in XML, attributes cannot have children.)

For convenience, a pattern that does not start with either / or // is implicitly assumed to start with //. This allows to just use an element name as ancestor pattern to match all elements of this name, as in DTDs.

Child Patterns: In its simplest form, a child pattern is a regular expression describing the content model of a set of elements. To allow some other features (e.g. groups) and not introducing ambiguity, all element names have to be prefixed with the keyword `element`. Regular expressions in

child patterns are built using concatenation (`,`), union (`|`), interleaving(`&`), Kleene closure (`*`), one-or-more (`+`), zero-or-one (`?`) and counting (`{n,m}`). The upper bound of counters may be `*` instead of a number to express that there is no upper bound. Sub-expressions can be grouped using round brackets. The use of the interleaving operator is restricted, to reflect the restrictions imposed by the all-pattern of XML Schema. (The restrictions for XML Schema are described in Section 3.8.2 in [12].) In plain words, these restrictions say that no content model should use an interleaving operator and at the same time a union or a concatenation operator. Furthermore, in content models containing an interleaving operator, counters are only allowed directly above element declarations in the syntax tree of the regular expression.

Priorities: It is possible to define BonXai rules such that two or more rules match the same path. When such a multiple match occurs, BonXai gives priority to the rule that occurs last in the schema. To illustrate, assume that we would change the ancestor pattern `content//section` to `section`. Then we would have the rules

```
section = mixed {attribute title,
                 (element section|group markup)*}
template//section = { element titlefont?,
                element style?, element section? }
```

in the schema. Both rules are matched by a `section` element that is below a `template` element. In cases like this, the rule that occurs last in the schema takes priority. Here, `template//section` takes priority and therefore the semantics of the modified schema are the same as the semantics of the original schema. The rationale behind priorities is that a developer can first write down rules that generally apply in the schema and write down the special cases and exceptions later. We introduced priorities in BonXai because they were required for ensuring full compatibility with XML Schema's expressive power. We explain priorities in more detail in Section 3.2.

Integrity Constraints, etc.: BonXai allows to express the same integrity constraints as XML Schema (i.e., unique, key, and keyref). The term "keyref" is taken from XML Schema, where it denotes a foreign key constraint.

BonXai's current implementation also models *attributes*, *groups*, *namespaces*, *mixed and nillable content models*, *references to foreign namespaces*, *wildcards*, and *annotations*.

3.2 Priorities in BonXai

In this subsection, we explain some fine points of the priority-based semantics of rules in BonXai schemas. Priorities were mainly introduced to avoid compatibility problems with XML Schema. However, we think they can also be convenient, as we will explain below.

In the theory of pattern-based schemas for XML (of which BonXai is an example), two alternative semantics for multiple matches of rules have been investigated [13, 16]: existential semantics and universal semantics. We say that the *ancestor-pattern of rule* `r = {s}` *matches a node* n in an XML tree, if the string of element names from the root of the document to n matches the regular expression `r`. The two semantics can now informally be defined as follows:

- Universal semantics: for each node n in the XML tree and each rule `r = {s}` for which the ancestor pattern matches n, the children of n must match `s`.
- Existential semantics: for each node n in the XML tree, there must be at least one rule `r = {s}` for which the ancestor pattern matches n and the children of n match `s`.

Thus, under universal semantics, we would require a matching element to match *all* content model definitions of relevant rules and under existential semantics, we would require a matching element to match *at least one* content model definition of a relevant rule.Unfortunately, neither semantics can be applied while retaining at the same time compatibility with the Unique Particle Attribution (UPA) rule of the W3C XML Schema specification [12, Section 3.8.6.4]. In a nutshell, UPA requires content model definitions to be *deterministic regular expressions* [4]. Furthermore, translating BonXai schemas under the universal or existential semantics to XSDs, requires deterministic regular expressions to be closed under finite unions and finite intersections, respectively, which is not the case [4, 6, 17]. As an aside we mention that deterministic regular expressions are not closed under complement.

A "quick and dirty" solution could be to require ancestor patterns in rules to have an empty intersection. However, we feel that this would be very user-unfriendly. Consider again our running example in Figure 5. The two ancestor patterns `template//section` and `content//section` have a non-empty intersection since both could, in theory, match a word that has an occurrence of `template`, followed by `content`, followed by `section` (even though such a word cannot occur as a path in trees defined by the schema). Changing the two ancestor patterns to mend this problem would make the schema less readable and require users to have deeper expertise in formal language theory.

We show in Section 4 that the priority-based semantics of BonXai does not have the expressivity problems of universal or existential semantics, by giving conversion algorithms from the core of BonXai to XML Schema and back; and by observing that the Unique Particle Attribution constraint is preserved. Furthermore, we feel that priorities make sense when designing schemas (specify general rules first, special cases later) and lead to more readable schemas. Therefore, a sensible way of using priorities is for cases where, for a set of elements with the same name, most of the elements have the same content model, but there are a few exceptions. (Notice that, if two ancestor patterns define regular expressions that end with different element names, the intersection of the rules is always empty and priorities are irrelevant.)

We conclude this section with a use case for priorities: schema evolution. In our running example, sections can be nested arbitrarily deeply. Assume that we want to change the schema such that the nesting depth of sections is at most three. In the BonXai schema in Figure 5, this can be achieved by inserting the rule

```
content/section/section/section =
                { attribute title, group markup }
```

at the end of the rules that start with `content`. The semantics of this rule would be that subsubsections only have a title attribute and markup, but no `section` children.

If one would want to perform the equivalent change directly in XML Schema, one would be required to make three complex types for sections below `content`: one for each allowed nesting depth. The change would introduce much more clutter.

3.3 A Comparison With Other Schema Languages for XML

As already stated before, BonXai borrows concepts from several existing schema languages for XML. The purpose of this section is to give an overview of the most well-known of those languages and discuss their relationship with BonXai.

Following [23], DSD2 [9] (Document Structure Description 2.0) is a language developed by the University of Aarhus and AT&T Research Labs whose primary goal is to be simple yet expressive. Like BonXai, DSD2 is based on rules which must be satisfied for every element in the input document. BonXai and DSD2 are incomparable in how context is defined. While DSD2 is far more expressive than DTDs, its exact expressiveness in formal language theoretic terms is unclear. It allows context to be defined in terms of Boolean expressions which can refer to structural predicates like *parent* and *ancestor*, but, unlike BonXai, also allows to look downward using predicates like *child* and *descendant*. BonXai on the other hand harnesses the full power of regular languages on the ancestor path, while DSD2 seems to remain within the star-free regular languages (on the ancestor path). For this reason, DSD2, on a structural level, is incomparable to XML Schema.

Relax NG [27] has been developed within the Organization for the Advancement of Structured Information Standards (OASIS). Like DSD2, its main goal is to combine simplicity with expressivity. In formal language theoretic terms, the expressiveness of Relax NG corresponds to the unranked regular tree languages which strictly includes XML Schema [24, 21]. Like XML Schema, Relax NG is grammar based and utilizes types to define context. However, Relax NG schemas are not restrained by the Unique Particle Attribution constraint or the Element Declarations Consistent constraint. So, unlike XSDs and therefore BonXai, the context of an element in Relax NG can depend on the complete tree. As BonXai strives for simplicity it utilizes a readable compact syntax which is inspired by that of Relax NG.

Schematron [28] is a rule-based language based on patterns, rules and assertions. Basically, an assertion is a pair (ϕ, m) where ϕ is an XPath expression and m an error message. The error message is displayed when ϕ fails. A rule groups various assertions together and defines by means of an XPath expression a context in which the grouped assertions are evaluated. Patterns then group various rules together. Schematron is not so much intended as a stand-alone schema language but can be used in cooperation with existing schema languages. BonXai shares the use of XPath-expressions with Schematron, although BonXai restricts them to a very small subset (linear expressions) to ensure compatibility with XML Schema.

Co-constraints is an overloaded term which generally refers to a mechanism for verifying data interdependencies. While DSD, Schematron, and Relax NG quite naturally allow to express co-constraints, XSDs are rather limited in this respect. The latter motivated the formulation of extensions of DTDs and XSDs, named DTD++ [11] and SchemaPath [7], with XPath expressions to express co-existence and co-absence of element names and attributes. These extensions share with BonXai the use of XPath to express conditions but differ from BonXai in that they increase the expressiveness beyond that of XML Schema.

4. THEORY: BONXAI VERSUS XSD

In this section, we explain the underlying theory behind BonXai. In particular, we provide

- a compact and clear formal model of core BonXai schemas, stripped of features that are unimportant for analysing conversion algorithms;
- a formal back and forth translation procedure between core XML Schema and core BonXai;
- an analysis of the blow-up of these conversions; and
- proof of worst-case optimality for the conversions.

Our aim is to provide a precise mathematical description of BonXai's core which abstracts away from unavoidable cosmetics like namespaces and data types, and which offers a quick understanding of the essentials of the language. The presentation of the translations between BonXai and XML Schema fulfills a similar purpose and, in addition, makes the relation between BonXai and XML Schema apparent. In particular, the translation provides insight to where one language can be more succinct than the other.

4.1 A Formal Model for BonXai Schemas

Before we introduce the formal model for the core of BonXai, we first establish some basic terminology and notation.

Basic Terminology.

We view an XML document as a finite, rooted, ordered, labeled, unranked tree D. We assume a finite alphabet (that is, a finite set) EName of *element names* from which the nodes of XML trees take their labels, that is, each node v of D carries exactly one label $\text{lab}(v) \in \mathsf{EName}$. By $a, b, c, \ldots$ we denote elements from EName. For a node v, we denote by $\text{anc-str}^D(v)$ the *ancestor-string* of v in D which is given by the concatenation of the labels of the nodes on the path from the root of D to v. More formally, the ancestor-string of v in D is the string $\text{lab}(v_1) \cdots \text{lab}(v_n)$, where v_1 is the root of D, $v_n = v$, and v_{i+1} is a child of v_i for each $i = 1, \ldots, n-1$. We denote by $\text{ch-str}^D(v)$ the concatenation of the labels of the children of v in D. More formally, if the children of v are $u_1, \ldots, u_m$ from left to right, then $\text{ch-str}^D(v) = \text{lab}(u_1) \cdots \text{lab}(u_m)$. We note that $\text{ch-str}^D(v)$ is sometimes also called the *content* of node v. We omit D in the notation of ancestor- or child-strings when D is clear from the context.

EXAMPLE 4.1. *Consider the* `section` *child v of the element* `template` *in the tree of Figure 1. Then*

$$\text{anc-str}(v) = \texttt{document template section}$$
$$\text{ch-str}(v) = \texttt{titlefont style section}.$$

We assume familiarity with finite automata and only discuss notation here. We denote a (nondeterministic) finite automaton or NFA as a tuple $A = (Q, \mathsf{EName}, \delta, q_0, F)$ where Q is its finite set of states, EName is the alphabet, $\delta : (Q \times \mathsf{EName}) \to 2^Q$ is the transition function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of accepting states. An NFA is *deterministic* if $\delta(q, a)$ contains at most one state for each $q \in Q$ and $a \in \mathsf{EName}$. The *language* of A (i.e., the set of words accepted by A) is defined in the standard manner. The *size* of A, denoted $|A|$, is the number of states of A. Sometimes we use finite automata without accepting states. We then simply write them as $A = (Q, \mathsf{EName}, \delta, q_0)$. We sometimes use $A(w)$ as an abbreviation the set of states that A can reach after reading w.

We use regular expressions r with the following syntax

$$r ::= \varepsilon \mid \emptyset \mid a \mid rr \mid r + r \mid (r)? \mid (r)^+ \mid (r)^*,$$

where ε denotes the empty string and a ranges over symbols in the alphabet EName. Sometimes we also use the symbol $\cdot$ for regular expression concatenation to improve readability. For a set $S = \{a_1, \ldots, a_n\} \subseteq \mathsf{EName}$ we sometimes abbreviate the disjunction $(a_1 + \cdots + a_n)$ by S. As usual, we write $L(r)$ for the language defined by regular expression r. We define the *size* of regular expression r to be its total number of alphabet symbol occurrences. For example, both expressions aaa and $a(b + c)?$ have size three.

A Formal Model for BonXai's Core.

Now we define *BonXai Schema Definitions (BXSDs)*, which are a formal model for the core of BonXai schemas. The difference between the BonXai schema specification language and BXSDs is that the former can be used in our implementation [19] and has most of the XML Schema Language features to make it usable in practice, whereas the latter is a stripped down version that we use here to study translations between BonXai and XML Schema. For instance, the BonXai language supports integrity constraints, but we do not define these in BXSDs since they are trivial to translate from and to XSD.

Our definition of BonXai Schema Definitions requires expressions s_i to be *deterministic* [4]. This restriction is necessary to make BXSDs expressively equivalent to XSDs, due to the UPA condition mentioned in Subsection 3.2. We note that such expressions are sometimes referred to as *one-unambiguous* [4]. We do not formally introduce deterministic regular expressions here, as BXSDs and XSDs have exactly the same restrictions and our conversion algorithms therefore do not alter the regular expressions of the content models. Given the lacking closure of DRE under Boolean operations noted in Subsection 3.2, it is however crucial, that none of the conversion algorithms presented in Subsection 4.2 construct unions, intersections, or complements of content models.

Definition 1. A *BonXai Schema Definition (BXSD)* is a pair $B = (\mathsf{EName}, S, R)$ where $S \subseteq \mathsf{EName}$ is a set of start elements and R is an ordered list $r_1 \to s_1, \ldots, r_n \to s_n$ of rules, where

- all r_i are regular expressions over EName and
- all s_i are deterministic regular expressions over the alphabet EName.

For each $i = 1, \ldots, n$, we say that the rule $r_i \to s_i$ has *index* i. Let D be an XML document and u a node of D. A rule $r_i \to s_i$ is *relevant* for u if i is the largest index such that anc-str$^D(u) \in L(r_i)$. Notice that a node u has at most one relevant rule in B. An XML document D conforms to the BXSD B if the label of root(D) is in S and, for each node $u \in$ Nodes(D), if $r_i \to s_i$ is relevant for u, then ch-str$^D(u) \in L(s_i)$. The definition of relevant rules reflects the priority system in BonXai: rules with a higher index have higher priority.

Example 4.2. *The formal abstraction of the BonXai schema in Figure 5 is the BXSD* $B = (\mathsf{EName}, S, R)$ *where*

- EName $= \{$*document, template, userstyles, content, section, style, title*$\}$
- $S = \{$*document*$\}$
- *R is the ordered list containing rules (parts omitted):*

$$\begin{aligned}
&//\mathit{document} &&\to \mathit{template\ userstyles\ content}\\
&//\mathit{content} &&\to \mathit{section}^*\\
&//\mathit{template} &&\to \mathit{section}\\
&//\mathit{userstyles} &&\to \mathit{style}^*\\
&//\mathit{content}//\mathit{section} &&\to (\mathit{bold} + \cdots + \mathit{section})^*\\
&\qquad\vdots\\
&//\mathit{template}//\mathit{section} &&\to \mathit{titlefont?\ style?\ section?}\\
&\qquad\vdots
\end{aligned}$$

Here, we wrote the left-hand-sides of BonXai rules as in Section 2. Formally, in this section, // abbreviates the regular expression EName*.

4.2 Translations Between Schemas

Before we discuss how to translate back and forth between XML Schema and BonXai, we give our abstraction of an XML Schema, closely following the definition from [24, 21, 20].

A Formal Model for Core XSDs.

An XML Schema uses a finite set of element names and complex type names. We therefore fix finite sets EName and Types of *element names* and *complex type names*, respectively. The set TEname of *typed element names* is then defined as $\{a[t] \mid a \in \mathsf{EName}, t \in \mathsf{Types}\}$. In an XML Schema, a typed element name $a[t]$ could, for example, be written as `<xs:element name="`a`" type="`t`"/>`.

Definition 2. An *XSchema Definition (XSD)* is a tuple $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ where EName and Types are finite sets of elements and types, respectively, ρ is mapping from Types to regular expressions over alphabet TEname, and $T_0 \subseteq \mathsf{TEname}$ is a set of typed start elements. Furthermore, the following two conditions hold:

Element Declarations Consistent (EDC) There are no typed elements $a[t_1]$ and $a[t_2]$ in a regular expression $\rho(t)$ with $t_1 \neq t_2$. Furthermore, there are no typed elements $a[t_1]$ and $a[t_2]$ in T_0 with $t_1 \neq t_2$.

Unique Particle Attribution (UPA) Each regular expression $\rho(t)$ is deterministic.

We sometimes also refer to $\rho(t)$ as the *content model associated to* t. The EDC constraint can be found in [12, Section 3.8.6.3] and the UPA constraint in [12, Section 3.8.6.4].

A *typing* of an XML document D w.r.t. X associates, to each node u of D, a type of the schema. Formally, a typing of D w.r.t. X is a mapping μ from Nodes(D) to TEname. A typing μ is *correct* if it satisfies the following three conditions:

- $\mu(\text{root}(D)) \in T_0$.
- For each node $u \in$ Nodes(D), we have $\mu(u) \in \{\text{lab}(u)[t] \mid t \in \mathsf{Types}\}$.
- For each node $u \in$ Nodes(D) with children $u_1, \ldots, u_n$ from left to right, we have $\mu(u_1) \cdots \mu(u_n) \in L(\mu(u))$.

An XML document D conforms to an XSD X if there exists a *correct typing μ of D w.r.t. X*. Notice that typings are unique due to the EDC condition, that is, there can be at most one correct typing for a given document D w.r.t. a given XSD X.

4.2.1 Translation from XML Schema to BonXai

We present a translation algorithm from XSDs to BXSDs. This algorithm is the core of a procedure that we implemented

to translate XML Schema into BonXai [19]. The algorithm consists of two phases. The first phase converts an XSD into an intermediate data structure, which is called a *DFA-based XSD*. We will define such a DFA-based XSD formally, because it is a representation of schemas that is very convenient in proofs. In the second phase, the DFA-based XSD is translated to the BXSD.

DFA-based XSDs were introduced in [20] (Definition 6) as an alternative characterization of XML Schema Definitions. We now define DFA-based XSDs as in [20], with a minor difference: due to the UPA condition, we require their content models to be deterministic regular expressions.

DEFINITION 3. A *DFA-based XSD (with deterministic content models)* is a tuple (A, S, λ), where $A = (Q, \mathsf{EName}, \delta, q_0)$ is a DFA with initial state q_0 and without final states such that q_0 has no incoming transitions, $S \subseteq \mathsf{EName}$ is the set of allowed root element names and λ is a function mapping each state in $Q \setminus \{q_0\}$ to a deterministic regular expression over EName. Furthermore, for every state $q \in Q$ and every element name a occurring in $\lambda(q)$, we have that $\delta(q, a)$ is non-empty.

In the remainder of the paper, S usually equals $\{a \mid \delta(q_0, a) \neq \emptyset\}$. (The intuition is that, for each element $a \in S$, the automaton A can read a string that starts with a. Since S is simply the set of root elements, λ does not map q_0 to a regular expression.) However, we sometimes use fully defined DFAs (which are DFAs in which $|\delta(q, a)| = 1$ for every state q and label a) and therefore we need to explicitly mention S in general. Since we *only* consider DFA-based XSDs with deterministic content models in this paper, we henceforth simply refer to them as *DFA-based XSDs*.

An XML document D *satisfies* (A, S, λ) if the root node is labelled with an element name from S and, for every node u, $A(\text{anc-str}^t(u)) = \{q\}$ implies that $\text{ch-str}^D(u)$ is in the language defined by $\lambda(q)$.

We now explain how to translate a given XSD $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ into an equivalent DFA-based XSD A in linear time. The procedure is outlined in Algorithm 1 and resembles procedures in [21, 13], which were developed for different models of XSDs.[3] It has the following property.

LEMMA 4 (ADAPTED FROM LEMMA 7 IN [13]). *Each XSD can be translated into an equivalent DFA-based XSD in linear time.*

We now show how to translate DFA-based XSDs into equivalent BXSDs. The translation is in Algorithm 2 and is similar to the proof of Theorem 7.1 ((a) ⇒ (d)) in [21].

LEMMA 5. *Each DFA-based XSD (A, S, λ) can be translated into an equivalent BXSD B with linearly many rules in $|A|$.*

Notice that the ordering of the rules in R in Algorithm 2 is arbitrary. The reason why the ordering is not important is that the priorities in BonXai are irrelevant in the schema. Indeed, for each pair of states $q_1 \neq q_2$ from A, we have that $L(r_{q_1}) \cap L(r_{q_2}) = \emptyset$, because A is a DFA. Furthermore, the BXSD B can have regular expressions that are exponentially larger than $|A|$ in general. This cannot be avoided[4] because A is a DFA and the worst-case conversion from a DFA to a regular expression is well-known to be exponential [10]. In Section 4.4 we discuss classes of schemas that capture most cases in practice and that do not lead to such a blow-up.

[3]One consequence of the slightly different models of XSDs is that the translation in [13] is quadratic, whereas it is linear in our case.

Algorithm 1 Translating an XSD to an equivalent DFA-based XSD.

Input: XSD $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$
Output: DFA-based XSD $(A = (Q, \mathsf{EName}, \delta, q_0), S, \lambda)$ equivalent to X

1: $S := \{a \mid \exists t \in \mathsf{Types} \text{ such that } a[t] \in T_0\}$
2: $Q := \{q_0\} \uplus \mathsf{Types}$
3: For each $a[t] \in T_0$, $\delta(q_0, a) := t$
4: For each $t_1 \in \mathsf{Types}$ and $a \in \mathsf{EName}$ such that $a[t_2]$ occurs in $\rho(t_1)$, $\delta(t_1, a) := t_2$
5: For each $t \in \mathsf{Types}$, $\lambda(t) := \mu(\rho(t))$
▷ $\mu(\rho(t))$ is obtained from $\rho(t)$ by replacing every $a[t']$ with a

Algorithm 2 Translating a DFA-based XSD into an equivalent BXSD.

Input: DFA-based XSD $(A = (Q, \mathsf{EName}, \delta, q_0), S, \lambda)$
Output: BXSD $B = (\mathsf{EName}, S, R)$ equivalent to X

1: **for** every state $q \in Q$ **do**
2: $r_q :=$ a reg. expression for $(Q, \mathsf{EName}, \delta, q_0, \{q\})$
3: $s_q := \lambda(q)$
4: $R := r_{q_1} \to s_{q_1}, \ldots, r_{q_n} \to s_{q_n}$, where $\{q_1, \ldots, q_n\} = Q$

4.2.2 Translation from BonXai to XML Schema

The translation from BonXai to XML Schema follows a similar overall outline as the reverse translation of Section 4.2.1. Again, we use DFA-based XSDs as an intermediate representation in the translation. That is, we first translate BXSDs into DFA-based XSDs and translate the latter to XSDs. However, the present translation is more technical than the one before.

Algorithm 3 describes the translation of BXSDs into DFA-based XSDs.

LEMMA 6. *Each BXSD B can be translated into an equivalent DFA-based XSD (A, S, λ) for which $|A|$ is at most exponential in $|B|$.*

It should be noted that Algorithm 3 is optimized for readability and not for efficiency. It is straightforward to change it such that it only computes reachable states of A. Note that whether a state is reachable also depends on the right-hand sides of the rules, because a transition $\delta(p, a)$, for which the label a does not occur in $\lambda(p)$, can never be taken in a conforming document.

The final translation we need is the one from DFA-based XSDs into XSDs. It is summarized in Algorithm 4 and has linear running time.

LEMMA 7 (ADAPTED FROM LEMMA 7 IN [13]). *Each DFA-based XSD can be translated into an equivalent XSD in linear time.*

We note that the XSD that results from Algorithm 4 can be "minimized" efficiently using a minor adaptation of the

[4]Proving that an exponential blow-up cannot be avoided is more technical than just this observation, see Section 4.3.

Algorithm 3 Translating a BXSD to an equivalent DFA-based XSD.

Input: BXSD $B = (\mathsf{EName}, S, R = r_1 \to s_1, \ldots, r_n \to s_n)$
Output: DFA-based XSD (A, S, λ) equivalent to B

1: **for** each $i = 1, \ldots, n$ **do**
2: $A_i :=$ minimal complete DFA $(Q_i, \mathsf{EName}, \delta_i, q_0^i, F_i)$ for $L(r_i)$
3: $A := A_1 \times \cdots \times A_n$ ▷ A has state set $Q_1 \times \cdots \times Q_n$
4: **for** each $(q_1, \ldots, q_n) \in Q_1 \times \cdots \times Q_n$ **do**
5: **if** $\exists i \in \{1, \ldots, n\}$ such that $q_i \in F_i$ **then**
6: $i :=$ largest number such that $q_i \in F_i$
7: $\lambda((q_1, \ldots, q_n)) := s_i$
8: **else**
9: $\lambda((q_1, \ldots, q_n)) := (\mathsf{EName})^*$

Algorithm 4 Translating a DFA-based XSD to an equivalent XSD.

Input: DFA-based XSD $(A = (Q, \mathsf{EName}, \delta, q_0), S, \lambda)$
Output: XSD $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ equivalent to (A, S, λ)

1: $\mathsf{Types} := Q$
2: $T_0 := \{a[\delta(q_0, a)] \mid a \in S, \delta(q_0, a) \neq \emptyset\}$
3: **for** each state $q \in Q$ **do**
4: $r_q :=$ expression obtained from $\lambda(q)$ by replacing each symbol a with $a[\delta(q, a)]$
5: $\rho(q) = r_q$

minimization algorithm for XSDs from [22]. (More formally, it is possible to efficiently produce an XSD such that the set Types is minimal among all equivalent XSDs. Also, the expressions r_q do not become larger.) The difference with the minimization algorithm from [22] would be that the deterministic regular expressions r_q should not be minimized. (In fact, it is not clear how to efficiently minimize a deterministic regular expression — if it would be possible to do this efficiently, the whole resulting XSD could be minimized in polynomial time by the algorithm from [22].)

4.3 Worst-Case Optimality of the Translation Algorithms

We now prove that both translation algorithms are worst-case optimal. In particular, we show that both conversions from the previous section can lead to exponential size blow-ups in general. In Section 4.4 we exhibit fragments that are prevalent in practice for which the conversions are efficient.

4.3.1 From XML Schema to BonXai

When converting an XML Schema (XSD) to a BonXai Schema Definition (BXSD) using the procedures in Lemmas 4 and 5 it is possible that the BXSD is exponentially larger than the XSD. The source of this exponential blow-up lies in Algorithm 2 which is used in Lemma 5. More precisely, line 1 constructs a regular expression equivalent to a DFA, which is well known to be exponential in the worst case [10].

We will now show that this blow-up cannot be avoided in general, which means that, in this sense, our conversion algorithm is worst-case optimal. Recall, however, that our conversion which we showed in Lemma 5 does not produce a large number of rules in the BXSD. Indeed, if the DFAs that Algorithm 2 encounters on line 2 only produce polynomially large regular expressions, then the whole conversion is polynomial as well. We discuss a particularly relevant such case in Section 4.4.

The following theorem is the most technical result in the paper. Its proof leverages a technique from [10]. The hard part of our proof is to show that the exponential blowup cannot be avoided by a clever use of the priorities in BonXai.

THEOREM 8. *There exists a family $(X_n)_{n\in\mathbb{N}}$ of XSDs such that, for each n, X_n has size $O(n^2)$ but the smallest BXSD equivalent to X_n has size at least $2^{\Omega(n)}$.*

PROOF SKETCH. Essentially, one needs to show that there exists a family of DFA-based XSDs $(X_n)_{n\in\mathbb{N}}$ such that every BXSD in which the left-hand-sides of rules reflect the DFA-types of the X_n requires exponential-size regular expressions. This means that we need to exhibit the existence of a family of DFAs such that the smallest equivalent regular expressions are necessarily exponential, even when they can exploit the limited negation of BonXai's priority system. To achieve this, we significantly extend and strengthen a technique of Ehrenfeucht and Zeiger [10] who showed that there is a class of languages $(Z_n)_{n\in\mathbb{N}}$, such that Z_n can be accepted by a DFA of size $O(n^2)$ but cannot be defined by a regular expression of size smaller than 2^{n-1}.

For every $n \in \mathbb{N}$ we let $\Sigma_n = \{a_{ij} \mid i, j \in \{1, \ldots, n\}\}$. We call i the *source* and j the *target* of a symbol a_{ij}. We define Z_n as

$$Z_n = \big\{w_1 \cdots w_m \in \Sigma_n^* \mid \forall i \in \{1, \ldots, m-1\}, \exists j, k, l \text{ such that } w_i w_{i+1} = a_{jk} a_{kl}\big\}.$$

That is, in every word in Z_n, the target of a symbol and the source of the following symbol must be equal. Every word $w \in \Sigma_n^* \setminus Z_n$ has a first symbol $a_{i\ell}$ whose target ℓ does not coincide with the source of the following symbol. We call ℓ the error index of w.

We now construct a family $(X_n)_{n\in\mathbb{N}}$ of XSDs, such that X_n is of size $O(n^2)$ and the smallest BXSD equivalent to X_n has size $2^{\Omega(n)}$. We define X_n by its DFA-based XSD (A_n, S_n, λ_n). To this end, we let $S_n = \Sigma_n$ and choose the components of $A_n = (Q \cup Q', \Sigma_n, \delta, q_1)$ as follows.

- $Q = \{q_i | 1 \leq i \leq n\}$ and $Q' = \{q_i' | 1 \leq i \leq n\}$;
- for every $q_i \in Q$ and $a_{j\ell} \in \Sigma$, $\delta(q_i, a_{j\ell}) = \begin{cases} q_\ell & \text{if } i = j \\ q_i' & \text{if } i \neq j \end{cases}$
- and, for every $q_i' \in Q'$ and $a_{j\ell} \in \Sigma$, $\delta(q_i', a_{j\ell}) = q_i'$,
- for every $q_i \in Q$, $\lambda(q_i) = \varepsilon \cup \Sigma$,
- for every $q_\ell' \in Q$, $\lambda(q_\ell') = \varepsilon \cup \Sigma \cup \{a_{\ell\ell} a_{\ell\ell}\}$.

In other words, A_n is a DFA that tests whether a word is in Z_n and remembers, for words not in Z_n, their error index.

The documents valid with respect to X_n are thus characterized by the following two properties.

- All label sequences over Σ_n are allowed in paths.
- The only allowed kind of branching is binary branching of the form $a_{ij} \to a_{\ell\ell} a_{\ell\ell}$ below nodes whose ancestor path contains a Z_n-error with error index ℓ.

We note that, as branching can only take place below an error, and the first error of a path is unique, in every document there can be binary branching $a_{\ell\ell} a_{\ell\ell}$ with at most one kind of symbols.

It is straightforward that X_n is of size $O(n^2)$. It can be shown that every equivalent BXSD B is of size $2^{\Omega(n)}$. □

4.3.2 From BonXai to XML Schema

We prove that the translation from BXSDs to XSDs is worst-case optimal.

THEOREM 9. *There exists a family of BXSDs $(B_n)_{n\in\mathbb{N}}$ such that, for each n, the BXSD B_n has size $O(n)$ but the smallest XSD equivalent to B_n has size at least 2^n.*

PROOF SKETCH. Let $n \in \mathbb{N}$ be arbitrary. Let $B_n = (\mathsf{EName}_n, S_n, R_n)$ be the BXSD with

$$\mathsf{EName}_n = \{a, a_1, \ldots, a_n, b_1, \ldots, b_n\},$$

$S_n = \{a_1, \ldots, a_n\}$, and R_n consisting of the following rules:

$$\begin{array}{rcl} //a & \to & \varepsilon \\ //(b_1 + \cdots + b_n) & \to & \varepsilon \\ //(a_1 + \cdots + a_n) & \to & (a + a_1 + \cdots + a_n) \\ //a_1//a_1//a & \to & b_1 \\ //a_2//a_2//a & \to & b_2 \\ \vdots & \vdots & \vdots \\ //a_n//a_n//a & \to & b_n \end{array}$$

Here we wrote the regular expressions on the left-hand-side of rules as in Section 2 with // as an abbreviation for EName^*. This schema defines a set of unary (i.e., non-branching) trees and its semantics is the following. If the ancestor path of an a-element contains, for each $1 \leq i \leq n$, at most one a_i element, its content model is ε. Otherwise, if j is the largest number such that a_j occurs at least two times on the path to the a element, then this a element has b_j as a child.

It can be proved with techniques from [22] that the smallest XSD equivalent to the above BXSD is exponentially large in n. Intuitively, in order to decide which b_i is the child under an a, the types of the XSD needs to keep track of the largest j, for which a_j has already occurred twice, and, worse, the set of $i > j$, for which a_i has already occurred once. □

4.4 Efficient Translations for Fragments

Even though the translations between XSD and BonXai in Sections 4.2.1 and 4.2.2 are provably optimal, they can be exponential in the worst case. In this section, we argue why we do not expect this to be a problem in practice. In particular, we prove that the translation is polynomial for a restriction of XSDs that accounts for the overwhelming majority of schemas in practice. An examination of 225 XSDs from the Web revealed that in more than 98% the content model of an element only depends on the label of the element itself, the label of its parent, and the label of its grandparent [21]. This motivates the study of the following class of DFA-based XSDs.

DEFINITION 10. A DFA-based XSD is *k-suffix*, if the type of an element only depends of the last k symbols of its ancestor string. More precisely, a DFA-based XSD (A, S, λ) with $A = (Q, \mathsf{EName}, \delta, q_0)$ is k-suffix based if $A(w_1 a_1 \cdots a_k) = A(w_2 a_1 \cdots a_k)$ for all strings w_1, w_2 over EName and symbols $a_1, \ldots, a_k \in \mathsf{EName}$.

Hence, 98% of the XSDs in the aforementioned study have a corresponding 3-suffix DFA-based XSD. Actually, this DFA-based XSD can be obtained simply by applying the construction of Lemma 4 to the given XSD. Furthermore, according to Lemmas 4 and 7, the translations between XSDs and DFA-based XSDs are straightforward and very efficient. We therefore do not revisit these constructions and focus on translations between (k-suffix) DFA-based XSDs and BXSDs. The BXSDs corresponding to this class of schemas can be defined as follows.

DEFINITION 11. A regular language L is a *suffix language* if $L = \{w\}$ or $L = L(\mathsf{EName}^* w)$ for some word w. It is a *k-suffix language* if, additionally, $|w| \leq k$. A BXSD (EName, S, R) is *k-suffix based* if, for every rule $r \to s$ in R, the left-hand side r is a k-suffix language.

The following theorem considers the translation from k-suffix based BXSDs and k-suffix DFA-based XSDs. It is similar in flavor to Proposition 5.2 in [16], but considers rules with a priority system as in BonXai. Kasneci and Schwentick avoided this issue by assuming that rules have pairwise disjoint left-hand-side languages.

THEOREM 12. *Each k-suffix based BXSD can be translated in polynomial time into an equivalent k-suffix DFA-based XSD of linear size.*

We now consider the reverse direction. An important difference with Theorem 12 is that this direction is exponential in k, that is, it needs k to be constant in order to be polynomial. However, as we noted before, in 98% of the schemas occurring in the practical study of [21], we see that $k \leq 3$.

THEOREM 13. *Let k be a constant. Each k-suffix DFA-based XSD can be translated in polynomial time into an equivalent k-suffix based BXSD.*

Finally, we note that it is easy to decide if a given XSD can be translated efficiently into a BXSD, i.e., whether it corresponds to a k-suffix DFA-based XSD (where k can either be fixed in advance or not). Questions of this kind were investigated in [8, 14, 26].

5. CONCLUSIONS

We introduced BonXai with the explicit goal of reconciling the expressivity of XML Schema with the simplicity of DTDs, thereby creating a de facto human-readable front-end for XSDs and providing a means to simplify XSD development. BonXai is a full-fledged schema language with many features and a formal specification [18]. The language can be employed in various scenarios (c.f., [19]) ranging from the creation of novel XSDs to debugging of existing XSDs. Furthermore, BonXai is built on a solid theoretical foundation which is rooted in pattern-based schemas [20, 21] and which facilitates transformation algorithms and their analysis. While transforming between BonXai and XML Schema can have high complexity in the worst case, our investigations show that for a very large and practically relevant class this is never the case. At the moment, BonXai cannot yet specify simple types natively. This means that in order to use simple types in BonXai one has to define them in an XSD which is then to be imported in the BonXai schema. Adding native support for simple type would probably be one of the most desirable extensions of the current language.

Acknowledgments

We acknowledge the financial support of grant number MA 4938/2–1 from the Deutsche Forschungsgemeinschaft (Emmy Noether Nachwuchsgruppe).

6. REFERENCES

[1] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML Schema: effortless handling of nondeterministic regular expressions. In *International Symposium on Management of Data (SIGMOD)*, pages 731–744, New York, NY, USA, 2009. ACM.

[2] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems*, 35(2):11:1–11:47, 2010.

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

[4] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[5] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[6] P. Caron, Y. Han, and L. Mignot. Generalized one-unambiguity. In *International Conference on Developments in Language Theory (DLT)*, pages 129–140, 2011.

[7] C. S. Coen, P. Marinelli, and F. Vitali. Schemapath, a minimal extension to XML Schema for conditional constraints. In *International World Wide Web Conference (WWW)*, pages 164–174, 2004.

[8] W. Czerwiński, W. Martens, and T. Masopust. Efficient separability of regular languages by subsequences and suffixes. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 150–161, 2013.

[9] DSD. Document structure description (DSD). `http://www.brics.dk/DSD/`, 2002.

[10] A. Ehrenfeucht and H. P. Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12(2):134–146, 1976.

[11] D. Fiorello, N. Gessa, P. Marinelli, and F. Vitali. DTD++ 2.0: Adding support for co-constraints. In *Extreme Markup Languages*, 2004.

[12] S. Gao, C. Sperberg-McQueen, H. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema definition language (XSD) 1.1 part 1: Structures. `www.w3.org/TR/2012/REC-xmlschema11-1-20120405/`, April 2012.

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

[14] P. Hofman and W. Martens. Separability by short subsequences and subwords. In *International Conference on Database Theory (ICDT)*, 2015.

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

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

[17] K. Losemann, W. Martens, and M. Niewerth. Descriptional complexity of deterministic regular expressions. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 643–654, 2012.

[18] 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 www.theoinf.uni-bayreuth.de/download/bonxai-spec.pdf, Manuscript 2014.

[19] W. Martens, F. Neven, M. Niewerth, and T. Schwentick. Developing and analyzing XSDs through bonXai. *International Conference on Very Large Data Bases (VLDB)*, 5(12):1994–1997, 2012.

[20] W. Martens, F. Neven, and T. Schwentick. Simple off the shelf abstractions of XML Schema. *Sigmod RECORD*, 36(3):15–22, 2007.

[21] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, 2006.

[22] W. Martens and J. Niehren. On the minimization of XML Schemas and tree automata for unranked trees. *Journal of Computer and System Sciences*, 73(4):550–583, 2007.

[23] A. Møller and M. Schwartzbach. *An introduction to XML and web technologies*. Addison-Wesley, 2006.

[24] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.

[25] D. Peterson, S. Gao, A. Malhotra, C. Sperberg-McQueen, H. Thompson, and P. Biron. W3C XML Schema definition language (XSD) 1.1 part 2: Datatypes. `www.w3.org/TR/2012/REC-xmlschema11-2-20120405/`, April 2012.

[26] T. Place, L. van Rooijen, and M. Zeitoun. Separating regular languages by piecewise testable and unambiguous languages. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 729–740, 2013.

[27] RelaxNG. Relax NG specification. `http://www.relaxng.org/spec-20011203.html`, 2001.

[28] Schematron. Schematron. `http://www.schematron.com/`, 1999.

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