BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema

Reference (Published version):
Martens, Wim; Neven, Frank; Niewerth, Matthias & Schwentick, Thomas(2017) BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema. In: ACM TRANSACTIONS ON DATABASE SYSTEMS, 42(3), p. 1-42 (Art N° 15)

DOI: 10.1145/3105960
Handle: http://hdl.handle.net/1942/24954

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

WIM MARTENS, University of Bayreuth
FRANK NEVEN, Hasselt University and Transnational University of Limburg
MATTHIAS NIEWERTH, University of Bayreuth
THOMAS SCHWENTICK, TU Dortmund University

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 Documents are notoriously difficult. In this article, we introduce the XML specification language BonXai which incorporates many features of XML Schema but is arguably almost as easy to use as 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 not to replace XML Schema, but rather to provide a simpler alternative for users who want to go beyond the expressiveness and features of DTD, but do not need the explicit use of types. Furthermore, XML Schema processing tools can be used as a back-end for BonXai, since BonXai can be automatically converted into 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 a formal model for a core fragment of BonXai and the translation algorithms to and from a core fragment of XML Schema. We prove that BonXai and XML Schema can be converted back-and-forth on the level of tree languages and we formally study the size trade-offs between the two languages.

ACM Reference format:
Wim Martens, Frank Neven, Matthias Niewerth, and Thomas Schwentick. 2017. BonXai: Combining the simplicity of DTD with the expressiveness of XML Schema. 1, 1, Article 1 (October 2017), 41 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Through its endorsement by the W3C, XML Schema [34] 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 [16]. In their book, Møller and Schwartzbach discuss the comprehensibility of XML Schema as follows [28, 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.*

From a theoretical perspective, the most significant change in the migration from DTDs to XML Schema is the introduction of *complex types*. This addition dramatically increases the expressiveness [26, 29], as explained next. In DTDs, the definition of an element is determined by the element's label. For instance, the DTD rule

```
<!ELEMENT section  (title, paragraph*)>
```

specifies that *every* `section` element should have the same type of content: a left child labeled `title`, followed by a sequence of children labeled `paragraph`. Complex types in XML Schema go beyond that and allow element definitions to depend on the context in which they appear (e.g., the

label of the parent or grandparent, or any other ancestor element).[1] For instance, in XML Schema we can define several types of `section` elements by using complex types. We could define that, if a `section` element is a child of a `preface` element, then it has the content as described before, but section elements appearing inside `chapter` elements are also allowed to have children labeled `subsection`.

XML Schema is currently at version 1.1. Throughout this article, whenever we do not explicitly mention a version number for XML Schema, we are referring to XML Schema 1.0. For the purpose of this article, XML Schema 1.1 can be understood as an extension of XML Schema 1.0, i.e., every valid XML Schema 1.0 Document is also a valid XML Schema 1.1 Document.

1.1 Complex Types in Practice

Surprisingly, early studies revealed that XML Schema Documents (henceforth, SDs) occurring in practice hardly took advantage of complex types to go beyond the expressivity of DTDs [2, 3]. 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 *(i)* that users did not know how to wield the extra expressiveness of XML Schema; *(ii)* that it was too cumbersome to write sophisticated and precise schemas when weighed against their obvious benefits or; *(iii)* that the full power of complex types is rarely needed in practice. Actually, Møller and Schwartzbach assert that the introduction of types is a major aspect complicating the design of XML Schema [28, p. 156]:

> *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 just too complicated for the majority of users. In fact, the practical study we discuss next emphasizes that structural constraints beyond the power of DTDs are still hardly used in practice.

Since the study of Bex et al. [2, 3] is over a decade old and had a corpus of only 225 SDs, we repeated this experiment on a much larger corpus of SDs. We were able to obtain 8080 unique, well-formed SDs from the Web and analysed to which extent they use ancestor information to determine complex types (see Section 4 for details). We observed the following: 80.66% of SDs associate at most one complex type to every element name. In other words, these SDs do not use complex types beyond the power of DTDs. For a further 13.03%, the complex types were determined by the parent context (as in the example before, with `preface` and `chapter`). A further 3.64% also took the grandparent context into account.

These studies are in strong contrast with the full expressive power of complex types in XML Schema 1.0 or 1.1. In their full power, XML Schema 1.0 complex types allow the type of an element in a document to depend on a regular condition on *all ancestors* of the element [26].[2] In practice, we therefore see that the power of regular languages on the entire path of ancestors is not used at all. Instead, in more than 97% of schemas, this regular test is extremely limited: it only looks at the element itself, its parent, and its grandparent. Since XML Schema 1.1, one can use *type alternatives* to allow attributes of ancestors influence the complex type of elements. We found 5 schemas (0.06%) that use this feature.

[1] Notice that, in principle, ancestors up to an arbitrary height *can* be relevant to determine the complex type of an element.
[2] We mean this in a very precise way, namely the equivalence between items (a) and (b) in [26, Theorem 7.1].

1.2 A Main Idea of BonXai

A main idea underlying BonXai is to make it easier to define schemas for which the content of elements depends on an easy-to-specify context. In BonXai, one would write the rule

```
section = {element title, (element paragraph)*}
```

to express the DTD rule before and the two rules

```
preface/section = {element title, (element paragraph)*}
chapter/section = {element title, (element paragraph)*, (element subsection)*}
```

to define the content of the `section` inside `preface`, resp. `chapter`, respectively. The semantics of the first of the two rules is "the content of every `section` element inside a `preface` element is a `title` element, followed by a sequence of `paragraph` elements" (similarly for the second rule).

```
<element name="preface">
  <sequence>
    [...]
    <element name="section" type="TprefaceSection"/>
    [...]
  </sequence>
</element>

<element name="chapter">
  <sequence>
    [...]
    <element name="section" type="TchapterSection"/>
    [...]
  </sequence>
</element>

<complexType name="TprefaceSection">
  <sequence>
    <element name="title" [...]/>
    <element name="paragraph" maxOccurs="unbounded" [...]/>
  </sequence>
</complexType>

<complexType name="TchapterSection">
  <sequence>
    <element name="title" [...]/>
    <element name="paragraph" maxOccurs="unbounded" [...]/>
    <element name="subsection" maxOccurs="unbounded" [...]/>
  </sequence>
</complexType>
```

Fig. 1. SD fragment with complex types. Some parts, which could, e.g., contain more element tags or complex type references, are omitted, indicated by `[...]`.

An SD fragment that uses complex types for expressing the `preface`- and `chapter` rules is shown in Figure 1. The semantics of this SD code is that "the `TprefaceSection` (resp., `TchapterSection`) complex types occur inside `preface` (resp., `chapter`) elements and the content of these types of sections are [as specified in the schema]". We note that one could also write the SD fragment using

anonymous complex types. We decided against this because anonymous complex types come with a few disadvantages and may be considered to be bad style [8]. We will discuss the relationship between XML Schema anonymous types and BonXai in Section 3.4.

We have implemented a prototype translation from BonXai to XML Schema 1.0. As such, (i) BonXai only contains features that can be translated to XML Schema and (ii) XML Schema tools can be used as a back-end for BonXai. We stress that the objective of BonXai is not to incorporate every feature that XML Schema offers. For instance, BonXai does not use named types and therefore does not support complex type inheritance (e.g. through restriction and extension). Complex type inheritance can be quite useful because it allows for a development style closely resembling object-oriented design and thereby facilitating modularization.

BonXai is rather intended to provide a way to specify and manipulate a large class of XML Schemas that only adds little additional complication beyond DTDs and is more human-readable than XML Schema. Many XML Schema features have a parallel in BonXai and many XML Schema documents can be translated to BonXai.[3]

The automatic translation into (and largely also 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 [13, 32, 33], each with their own user base, they cannot be directly compiled into XML Schema for the simple reason that they can express schemas that are not definable in XML Schema 1.0 or 1.1. We give a comparison with contemporary schema languages in Section 3.5. On the theoretical side, a strength of BonXai is its solid foundation which is rooted in pattern-based schemas [25, 26] and which facilitates reasoning and transformation algorithms [17, 20].

1.3 Structure of the Article

The focus of this article is on the main features of BonXai and the mathematical foundations of the back-and-forth translation algorithms between BonXai and XML Schema. The paper is structured as follows.

Section 2 provides a light-weight introduction to BonXai. In particular, we present an example document together with a DTD, we show how the DTD can be written in BonXai and, subsequently, how the schema can be extended with features that go beyond the capabilities of DTDs. (An equivalent SD to the final BonXai schema can be found in the Appendix.)

In Section 3, we discuss more features of BonXai, their correspondence to XML Schema features, and consider its relationship to other XML schema languages. We also discuss our prototype implementation for a translator from DTD to BonXai and from BonXai to XML Schema and back (available at www.bonxai.org). Section 4 contains details on the practical study we conducted on SDs.

In Section 5, we present the formal core of the translation routines between BonXai and XML Schema. To this end, we define formal models for the theoretical core of XML Schema and BonXai, stripped of all the features that are not essential for understanding the translations. We bring both languages to the level of abstraction where they define sets of labeled unranked trees. We formally describe the translation algorithms on this level and prove that, in terms of languages of labeled unranked trees, BonXai is equally expressive as XML Schema. Furthermore, we analyze the worst-case blow-ups in these translations and prove why our algorithms are worst-case optimal.

[3]From our corpus, 2220 schemas (27,5%) only use features that we already support in BonXai and for which our translation to BonXai is exact (for other schemas, the translation to BonXai may approximate). If we would add native support for simple types, this number would be 5621 (69.6%).

```
<document xmlns="http://example.org">
  <template>
    <section>
      <titlefont name="SomeFont" size="42"/>
      <style><font name="Times" size="12"/></style>
      <section>
        <titlefont size="23"/>
      </section>
    </section>
  </template>
  <userstyles>
    <style name="userdefined1">
      <font name="MyFancyFont"/>
      <color color="red"/>
    </style>
    <style name="...">
      ...
    </style>
  </userstyles>
  <content>
    <section title="Introduction">
      In this paper we discuss ...
      <section title="Motivation">
        Our problem is important because ...
        <bold>This text is bold</bold><italic>and this is italic</italic>
        <style name="userdefined1">
          This text is red and uses a different font.
        </style>
      </section>
    </section>
    <section title="...">
      ...
    </section>
  </content>
</document>
```

Fig. 2. An XML document for our running example.

Finally, we discuss practically relevant fragments of XML- and BonXai Schemas in which the conversions are particularly efficient. We conclude in Section 6.

2 BONXAI BY EXAMPLE

In this section, we discuss a detailed (but still toy) example to highlight several features of BonXai. We will follow the use case of an existing DTD in which we want to incorporate XML Schema features.

Example 2.1 (An example document). Consider the XML document in Figure 2 (and its associated tree representation in Figure 3). The document is intended to represent content formatted in a fictional markup language. The `document` element has three children: `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.

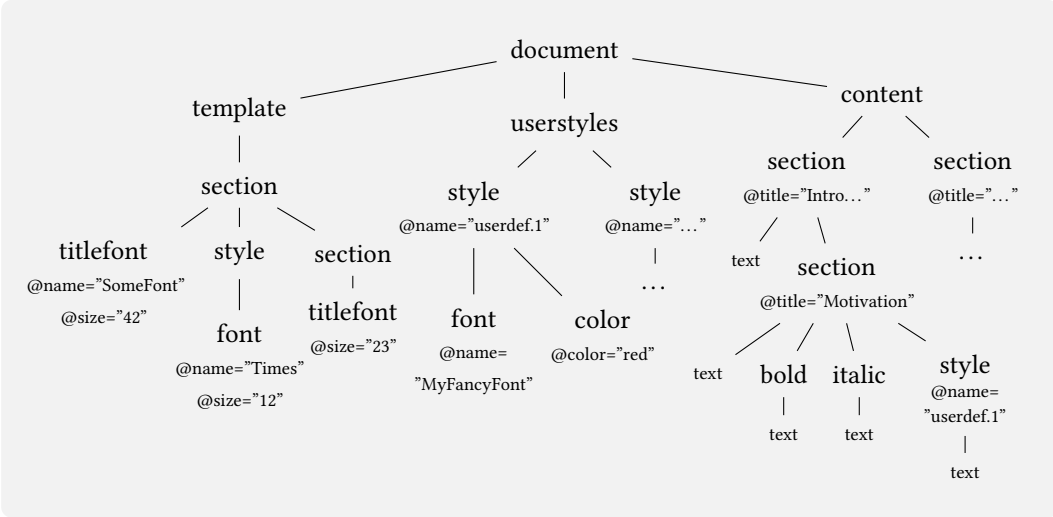

Fig. 3. Our example XML document as a tree.

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 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 syntax and semantics, notably, the `section` element. Similarly, `style` has a different semantics if it is used within `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). Figure 4 contains a DTD against which the XML document from Figure 2 is valid. We use the entity `markup` that allows us to write the schema more succinctly. Notice that all element names in `markup` can be used recursively. This is because we did not want to specify a specific ordering on how markup such as `bold`, `italic`, etc. should be nested in the tree.

We now discuss two BonXai schemas for the running example. The BonXai schema in Figure 5 closely resembles the DTD given in Figure 4 and, in particular, does not use context information beyond the power of DTDs. The BonXai schema in Figure 6 on the other hand uses this additional

```
<!ENTITY  % markup      "bold | italic | font | style | color">
<!ELEMENT   document    (template, userstyles, content)>
<!ELEMENT   template    (section)>
<!ELEMENT   userstyles  (style)*>
<!ELEMENT   content     (section)*>
<!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>
```

Fig. 4. A DTD describing the XML document in Figure 2.

expressiveness, in the same way as an SD would. Both examples use a compact syntax inspired by Relax NG [32].

We first discuss the BonXai schema in Figure 5 and draw parallels with the DTD. In fact, the only semantical differences between this BonXai schema and the DTD are that the BonXai schema has a target namespace, it defines a global element (`document`) that can be referred to from outside, and it uses the XML Schema simple types `string` and `integer`.

The main part of the BonXai schema is inside the `grammar` block. Like a DTD, this block consists of a collection of rules. The right-hand sides of rules denote content model definitions. In this example, the left-hand sides are just element- or attribute names.[4] Comparing the DTD and the BonXai schema, the only essential difference is that, in BonXai, we can immediately specify the attributes of an element in its content model definition, similarly as in XML Schema.

The BonXai schema in Figure 5 is written to make the correspondence to the DTD apparent, but it could be written more compactly. For instance, using regular expressions over element names on the left-hand sides of rules, we could just as well write:

```
(bold | italic) = mixed { (group markup)* }
```

The semantics of this rule would be that "all `bold` or `italic` elements have mixed content and allow all elements inside the `markup` group as children". We see here that, unlike anonymous types in XML Schema, the content model definitions for one element (`bold`) can be re-used for defining the content of another element (`italic`). Similarly, we could also write a single rule for the attributes `name`, `color`, and `size`:

```
(@name | @color | @title) = { type xs:string }
```

Next, we want to illustrate how BonXai's expressiveness can be used to go beyond the type mechanism of DTDs. In our example, we actually have two types of sections: one is used within `template` and the latter one within `content`. The former contains definitions for formatting, while the latter has the actual text content of the document. The rule

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

[4] In general, the left-hand sides can be arbitrary regular expressions over element names (optionally ending with attributes).

```
target namespace http://example.org
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 }
}
```

Fig. 5. A BonXai schema similar in spirit to the DTD in Figure 4.

defines the former kind. It stipulates that `section` elements occurring somewhere below a `template` element can contain a `titlefont` child, a `style` child, and a `section` child. Left-hand sides of BonXai rules can use the XPath axes `/` and `//`, which stand for "child" and "descendant", respectively. The latter kind of section can be defined with the rule

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

which stipulates that elements occurring somewhere below a `content` element should contain a title attribute, optional further sections and markup, and may contain text (indicated by the keyword `mixed`). The keyword `mixed` allows mixed content, i.e., it is allowed to interleave text with XML tags. A full BonXai schema is defined in Figure 6. (For completeness, we provide an XML Schema equivalent to the BonXai schema in Figure 6 in Appendix A.)

Left-hand sides of BonXai rules can use arbitrary regular expressions over element names. This gives BonXai, in theory, the same power to define element content depending on its ancestors as complex types in XML Schema 1.0. We prove this in Section 5 by giving back and forth translations between a formalization of a core of BonXai and a core of XML Schema 1.0.[5] In the regular expressions over element names, BonXai denotes concatenation, disjunction, Kleene star, and "optional" by "`,`", "`|`", "`*`", and "`?`", as in DTDs. The operator "`&`" stands for unordered concatenation, which is known as `xs:all` in XML Schema.

[5]Expanding BonXai with a feature that corresponds to XML Schema 1.1 *type alternatives* is conceptually not difficult. The main idea of the proofs in Section 5 would be the same, but their precise formulation would be burdened with the additional incorporation of attributes.

```
target namespace http://example.org
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 }
}
```

Fig. 6. A BonXai schema equivalent that describes the document in Figure 2 more precisely and uses several XML Schema features. An XML Schema describing the same set of XML documents as this BonXai schema can be found in Appendix A.

A left-hand side of a BonXai rule starting with / means that the matching of the left-hand-side must start at the root of the document. For instance, we could just as well have written `/document`. A left-hand side starting with // allows the first element to match anywhere in the document. If a left-hand side does not start with / or //, we implicitly assume that it starts with //.

The main difference between the BonXai schema in Figure 6 and the XML Schema Definition in Appendix A is that contexts in BonXai are defined explicitly. Another way of viewing the difference between XML Schema and BonXai is top-down versus bottom-up. In XML Schema, all relevant information about the root-path is propagated in a top-down fashion, encoded in types, while BonXai, instead, looks upward from a node, thus separating types from their inference.

3 BONXAI, THE PRACTICAL LANGUAGE

In Section 3.1, we present BonXai in more detail but do not intend to discuss every feature of the language. Instead, we provide a high-level overview and refer the reader to [22] for further details. We just discuss a few BonXai-specific matters (ancestor patterns, child patterns, and priorities) and, since we want to translate BonXai to XML Schema, argue how BonXai seamlessly incorporates many features also present in XML Schema (like differentiation between elements/attributes, element- and attribute groups, namespaces, constraints, schema imports, mixed types, default

values, anytype/anyattribute). Section 3.3 explains BonXai's priority system and Section 3.4 the relationship between BonXai and XML Schema's anonymous complex types.

The design of BonXai is heavily influenced by existing XML schema languages. We discuss these in Section 3.5. In Section 3.6, we discuss a system we built for developing and working with BonXai schemas and for translating back and forth between BonXai and XML Schema. In Section 3.7 we discuss a selection of use cases for the system.

3.1 The BonXai Schema Specification Language

BonXai schemas consist of up to five blocks. First is 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, we have an optional *group block*, which can define groups, similar to XML Schema groups (but without type information). 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.

We now discuss selected constituents of BonXai and their relationship to XML Schema. We make two groups: *BonXai-specific constructs*, which are constituents for which BonXai differs from XML Schema, and *compact syntax constructs*, which have exactly the same semantics in XML Schema than in BonXai.

3.1.1 BonXai-Specific Constructs. The following constructs constitute the core of BonXai. Here we explain them from a practical point of view. We do not present their formal semantics in detail, which is out of scope of the article. However, Section 5 formally defines BXSDs, which are very close to the restriction of BonXai to these constructs.[6]

Global Elements. These are the elements declared in the `global` block. Global elements can occur as root elements in XML documents that match the schema. Furthermore, global elements are precisely the elements that can be referenced from foreign namespaces. In our running example, there is a single such element, called `document`.

Global elements in BonXai therefore fulfil a similar role as they do in XML Schema. In XML Schema, the definition of global elements is implicit: an element is global if and only if it is defined directly below the xs:schema element. We chose to have an explicit definition because this allows us to write simpler rules in the `grammar` block. Indeed, if a rule in the grammar block just has an element name as a left-hand side, this does not automatically imply that this element is global. As such, the explicit global block allows rules in the grammar block to look more like DTD rules.

Rules. Rules within the grammar-block of a BonXai schema are of the form

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

Intuitively, the semantics of such a rule is that "every node selected by the ancestor pattern should have a list of children that satisfies the child pattern".

Ancestor Patterns. The ancestor pattern (left of the equality sign in rules) 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

[6]The formal model in Section 5 ignores features that are not essential for understanding the translation between BonXai and XML Schema, such as attributes, namespaces, the special syntax for ancestor patterns, and some of the operators in child patterns.

or child), "//" (descendant), "*" (Kleene star), "+" (one-or-more), and "?" (zero-or-one). Subpatterns can be grouped using round brackets.

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. We allow ancestor patterns to contain attribute names (prefixed with @) for specifying the types of the attributes.

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 counters (`{n,m}`). We may write `{n,*}` to indicate that `m` is unbounded. Subexpressions can be grouped using round brackets. The interleaving operator `&` is BonXai's equivalent of the all-pattern in XML Schema. As such, its use is restricted in the same way as all-patterns are restricted in XML Schema, see [16, Section 3.8.2]. (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.)

Alternative to being a regular expression, the child pattern can also be a type reference to some type specified in some (foreign) namespace. This is mostly useful for using simple types. For more details, we refer to the paragraph about type references further in this section.

In the case that the ancestor pattern of the rule can match an attribute, i.e., it contains a label starting with `@`, the child pattern has to be a reference to a simple type.

The type of the attribute can either be declared directly in the child pattern containing the attribute, or by a separate rule that matches the attribute, i.e., whose ancestor pattern selects the attribute.

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 in the running example of Section 2 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 that BonXai can be translated into XML Schema. We explain this matter in more detail in Section 3.3.

3.1.2 Compact Syntax Constructs. The semantics of the following constructs is the same in BonXai and in XML Schema. BonXai simply offers a compact syntax for them. As a consequence, translating these back and forth between BonXai and XML Schema is not difficult.

Attributes. Attributes are specified at the beginning of child patterns, that is, child patterns can have an optional list of attribute declarations before the start of the element declarations. Attributes are seperated by comma and can be followed by a ?, indicating that the attribute is optional.

Groups. Groups can be used in BonXai to abbreviate parts of child patterns that are common to several different patterns, similar to internal entities in DTD and groups in XML Schema (although BonXai groups do not have complex type information). In our running example, we use the group `markup`, to abbreviate the disjunction of the elements `bold`, `italic`, etc. Groups are declared in the groups block and can be used using the keyword `group` inside child patterns.

Attribute groups can be used analogously to groups. They are prefixed by the keyword `attribute-group`, as for example in the rules for `font`-elements in Figure 6.

Mixed and nillable content models. Mixed or nillable content models are declared using the keyword `mixed`, respectively, `nillable`, in front of the child pattern. Both keywords can be combined.

Default and fixed values. Default values for attributes and elements using a simple type can be declared using the syntax `type <typename> default "<value>"`. Fixed values can be declared analogously.

Integrity Constraints. 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. As in XML Schema, keys should have a name, so that keyrefs can refer to them. The general syntax of key constraints is

```
key <name> <ancestor pattern> { <selector> { <fields> } },
```

where the ancestor pattern is used to select the context for which the key should be defined and selector and fields have the same meaning as in XML Schema. The syntax for unique constraints is the same. The semantical difference between key and unique is – as in XML Schema – that a key requires all fields to exist, which is not required for unique constraints.

In a keyref, the semantics of `<name>` is that it should be the name of the key it refers to.

Example 3.1 (Keys for Example 2.1). To express in our running example that names of user-defined styles be unique, we can use the key constraint

```
key stylekey /document { //userstyles/style { @name } }
```

It says that, below the `document` root, paths that match `//userstyles/style/@name` uniquely identify paths that match `//userstyles/style` (as in XML Schema).

Finally, we can express that every `style` used in `content` should be declared in `userstyles` by the foreign key constraint

```
keyref stylekey /document { //style { @name } }
```

Namespaces. BonXai has full namespace support. The target namespace is declared using the keyword `target namespace`. Other namespaces can be declared using the syntax `namespace <prefix> = <namespace URI>`. The target namespace will be used as default namespace for all names, which are not prefixed with a namespace prefix. Names in other namespaces can be expressed by `<prefix>:<local name>`, as in XML Schema.

To reflect the usual practice of using qualified element names and unqualified attributes, all element names without a namespace prefix are in the target namespace and all attribute names without a namespace prefix are in the empty namespace, i.e., unqualified.

It is possible (but discouraged) to use unqualified element names by binding the empty namespace to some prefix. Qualified attribute names can be achieved by adding a namespace prefix for the target namespace and explicitly using this namespace prefix for all attribute names that should be qualified.

References to foreign namespaces. BonXai allows to refer to content of foreign XML Schemas, so that content that is defined elsewhere does not need to be re-defined within the BonXai schema.

In particular, it is possible to refer to foreign elements, attributes, and XML Schema simple- or complex types. We explain how foreign content can be referenced and how we intend the use of foreign references in BonXai.

Global elements of foreign namespaces can be referenced by using the `elementref` keyword inside child patterns. (In XML Schema it is only possible to refer to foreign elements if they are global elements in the foreign schema. We inherit this restriction.) For example, if we want to be able to embed SVG vector images, this can be accomplished in our running example by adding the namespace declaration `namespace svg=http://www.w3.org/2000/svg` and extending the group `markup` with `elementref svg:svg`.

Similarly, foreign global attributes can be referenced by using the `attributeref` keyword. For example, if we want to be able to add XLink[7] references to documents, this can be accomplished by adding the namespace declaration `namespace xlink=http://www.w3.org/1999/xlink` and extending the content model of the `document` rule with `attributeref xlink:href?`. (We explain how to import a bigger fragment of the XLink language when we discuss wildcards next.)

Type References. References to types (in foreign XML namespaces) are mainly intended to refer to simple types like `xs:integer` and `xs:string`. Type references are expressed by replacing the right-hand side of a rule with `{ type ns:typename }`, where `type` is a keyword, `ns` should be a declared namespace, and `typename` the name of the target type inside namespace `ns`. In our running example, the rule `@title = { type xs:string }` express that all `title` attributes throughout the document should use the type `string`, which is declared in the XML Schema namespace.

In general, it is also possible (but perhaps not encouraged) to refer to foreign XML Schema complex types. For example, the rule `//foo = { type svg:svgType }` would state that each element with name `foo` has the type `svgType` of the `svg` namespace. (However, we feel that using `elementref svg:svg` instead, whenever possible, is more elegant.)

Right now, all type references need to go to a foreign namespace, as BonXai does not allow users to define new types. (We consider extending BonXai with a syntax for defining simple types, see Section 3.2.)

In summary, although BonXai is intended to be a language that reduces the use of types to a minimum, we do allow references to foreign types. The reasons for this decision are that it allows the use of XML Schema simple types and that we like to allow users to easily import (e.g., well-known, standard) types which are defined elsewhere. It should be noted that, whenever an element is declared to have an (XML Schema-)type, no BonXai rules are applied to nodes below this element, as the set of allowed subtrees for this element is entirely determined by the type.

Wildcards. Wildcards are expressed by any-patterns in XML Schema. Note that XML Schema wildcards can be restricted to certain namespaces and it can be declared whether elements matched by any-patterns should be checked against some schema declaration or not. BonXai provides the same mechanism for wildcards. For example, to allow arbitrary foreign markup, we could extend the `markup` group with `any {lax namespace {##other}}`, meaning, that elements from other namespaces are allowed and should be validated, if a declaration is present. As in XML schema, the validation policy can be changed to `strict` (a declaration has to be present) or `skip` (the subtree below matched elements is not validated at all).

It is also possible to allow arbitrary attributes using the keyword `anyattribute`. As for arbitrary elements, the wildcard can be restricted to certain namespaces. For example to allow arbitrary XLink information to be added to document roots, we can extend the `document` node by `anyattribute`

[7]XLink is a language intended to allow embedding of hyperlinks and some other meta-information to arbitrary XML documents in a standardized way.

`{strict namespace {http://www.w3.org/1999/xlink}}`. The `strict` keyword says that the content should be validated and validation should fail if the XLink declaration is not present.

Annotations. Annotations can be used to add further information to a schema. In BonXai, annotations can be added before every rule. Annotations have no semantical meaning for the schema. However they might have a meaning for software used to create and edit BonXai schemas.

Our implementation (see Sections 3.6 and 3.7) uses annotations to preserve type names when converting XML Schema to BonXai. This way the user can easily grasp the correspondence between XML Schema complex types and BonXai rules. When converting BonXai to XML Schema, these annotations are used to generate meaningful XML Schema complex type names. For example, our implementation uses the annotation

```
@typename=MyTypename
//a = { ... }
```

with the meaning that a complex type created for the rule `//a = { ... }` should be named `MyTypename` when converting to XML Schema. In theory, it may be possible that more than one XML Schema complex type needs to be created for a single BonXai rule. In this case our implementation adds numbers after the given name.

Unconstrained Elements. It is theoretically possible to write BonXai schemas which do not constrain certain ancestor paths. For example, if a BonXai schema only has the two rules

```
/a = { element b, element c}
//b = { ... }
```

then the *c*-child of the root in a corresponding document does not have a matching ancestor pattern. In this case, BonXai allows any content below this `c`-child. Concretely, we translate this case to XML Schema's *anytype*, which is the most general type in XML Schema [30, Section 3]. We treat such elements the same as elements that refer to an XML Schema type (see the last paragraph of *References to foreign namespaces*). Therefore, as a consequence, no BonXai rules are matched against descendants of the `c`-child of the root.

3.2 Future Extensions

We briefly mention some possible future extensions of BonXai.

Simple Types. A compact syntax for specifying simple types is a natural extension of the BonXai language. There are two main reasons why simple type specifications were not there from the beginning. The first is that we wanted to concentrate first on the core of the language: structural expressiveness and the rule mechanism. The second reason is that there is a workaround for those who want to use new simple types in a BonXai schema: one can define an XML Schema that only defines these simple types and import it in BonXai. For these reasons, simple types were not a priority.

Type Alternatives. XML Schema 1.1 allows to specify alternative types for elements, where the effective type is selected by some attribute value of the instance document.

If the need arises, it is possible to add support for alternative content models to BonXai by allowing XPath node tests in ancestor patterns. These node tests can be restricted in a syntactical way to keep compatibility with XML Schema, i.e., to precisely allow those rules that can be mapped to XML Schema using the type alternatives mechanism.

Substitution Groups. Support for substitution groups can, in principle, be added to BonXai. However, we do not know yet if this extension makes sense since, in practice, substitution groups are very often used in combination with complex type inheritance, which we do not support.

3.3 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 [17, 20]: 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 [16, Section 3.8.6.4]. In a nutshell, UPA requires content model definitions to be *deterministic regular expressions* [6]. Furthermore, translating BonXai schemas under the universal or existential semantics to XML Schema, requires deterministic regular expressions to be closed under finite unions and finite intersections, respectively, which is not the case [6, 9, 21]. As an aside we note that deterministic regular expressions are also not closed under complement [9, 21].

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 6. 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 5 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 6, 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.

3.4 BonXai and XML Schema Anonymous Types

We briefly discuss the relationship between XML Schema anonymous types and BonXai. As BonXai does not explicitly use named types, one may believe that it essentially offers a compact syntax for anonymous complex types in XML Schema. However, this is not the case for the simple reason that BonXai is more expressive. For instance, BonXai allows the content of nodes to depend on ancestors arbitrarily high in trees (cfr. Section 5) which is not possible using only anonymous types.

From a more practical perspective, XML Schema anonymous types come with a set of disadvantages and may be considered to be bad style. Butek and Kendrick [8] note the following disadvantages of anonymous types:

(1) they cannot be re-used to define content of other elements;
(2) they still often must be named (as some applications need names for the complex types); and
(3) they may render a schema less readable (they are less elegant as named types).

In contrast, BonXai does not have any of these three disadvantages:

(1) BonXai content model definitions *can* be re-used for other elements (see, for example, in Figure 6, where we use the same content for `bold` and `italic` elements or for `font` and `titlefont` elements).
(2) BonXai uses XML Schema as a back-end and our translation[8] from BonXai to XML Schema does not generate anonymous types. As explained in Section 3.1 (under *Annotations*), users can even specify the name of the complex type to be generated.
(3) This is purely a matter of opinion, but we feel that BonXai rules are even simpler to understand than XML Schema code with named types. For instance, we believe that the two BonXai rules for `preface/section` and `chapter/section` in the Introduction are simpler to understand than the corresponding fragment of the XML Schema document.

3.5 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 [28], DSD2 [13] (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 [32] 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

[8]Here, we refer to the translation we implemented, not the translation between the formal models of the core of the languages in Section 5, which is a drastic simplification.

Table 1. Properties and Features of schema languages

	DTD	BonXai	XML Schema	Relax NG	DSD2	Schematron
convertible to XML Schema[a]	●	●	●			
rule based	●[b]	●			●	●
grammar based	●[b]		●	●		
standalone schema language	●	●	●	●	●	○[c]
compact syntax	●	●		●		
namespace support		●	●	●	●	●
integrity constraints	○[d]	●	●	○[d]	●	●
type hierachies			●			
simple types		○[e]	●	○[e]	●	○[f]

[a] For every schema there exists an XML Schema document that accepts the same set of XML document.
[b] Due to their simplicity, DTDs can be seen as both rule- and grammar based.
[c] Possible to use stand-alone, but usually used as supplement to a grammar based language.
[d] Very limited constraint support by ID/IDREF mechanism.
[e] Usage of externally defined simple types is possible.
[f] Simple types can be emulated using assertions.

regular tree languages which strictly includes XML Schema [26, 29]. 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 XML Schema 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 [33] 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, XML Schema is rather limited in this respect. The latter motivated the formulation of extensions of DTDs and XML Schema, named DTD++ [15] and SchemaPath [10], 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.

We composed an overview of a set of properties (top half) and features (bottom half) of the languages we discussed here in Table 1. A bullet means that we believe the language to have the respective property (or support the respective feature). Compared with other schema languages, the novelty of BonXai is that it

(1) can be translated automatically into XML Schema and
(2) offers a novel compact syntax that makes the relationship between the content of elements and context information explicit.

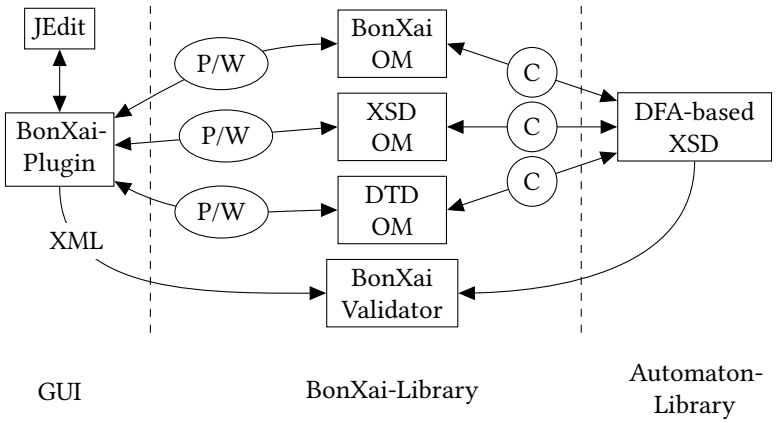

Fig. 7. Schematic overview of the system components. P/W: Parser/Writer, C: Converter, OM: Object Model.

Item (1) implies that BonXai can make use of the wide array of tools for XML Schema such as validators, mappers into programming languages, query optimization engines that take XML Schema information into account, etc. DSD2, Relax NG, and Schematron can express schemas that are not expressible as XML Schema[9] and therefore do not have this feature. Concerning item (2), we note that although Relax NG has a compact syntax, it also does not make the relationship between content and context of elements explicit. Compared to XML Schema, we feel that BonXai's compact syntax renders many schema definitions more readable, as argued in the Introduction and in Section 2.

3.6 Implementation and System

We implemented a prototype system that supports the development and maintenance of BonXai schemas, which was presented at VLDB 2012 [23] and is available at www.bonxai.org. The system includes a parser for the BonXai language, conversion routines (between BonXai and XML Schema, from DTD to BonXai), and native validation against BonXai schemas.

Figure 7 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.

3.6.1 User Interface. The GUI of our current implementation is provided through a plugin for the open source editor JEdit [19]. JEdit provides basic text editing functionalities, syntax highlighting and a flexible plugin interface. Through the GUI, the user can directly develop BonXai schemas. The BonXai-Plugin provides the connection between the BonXai library and JEdit. We also developed a command line client which provides an easy way for batch conversion of schemas.

3.6.2 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 XML Schema inference algorithms [4] or algorithms for repairing the unique particle attribution constraint of XML Schema [1]. The automaton library is capable of storing so-called *DFA-based XSDs* [25], which are an intermediate format between XML Schema and BonXai and which we extended

[9]This can be proved using [26, Theorem 7.1].

to be able to cope with XML Schema features (for instance, those discussed in Section 3.1.2). We discuss the underlying principles of DFA-based XSDs in more depth in Section 5 of this article.

3.6.3 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, XML Schema, and BonXai. It also supports native validation of XML against DFA-based XSDs. The theoretical foundations of the conversion algorithms between BonXai and XML Schema are discussed in Section 5.

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

Conversion Routines. As all conversions pass through the DFA-based XSD representation, there are six conversion routines. The translation to and from XML Schema and DTD is rather direct (not difficult from a theoretical perspective). The computation of a DFA-based XSD from a BonXai schema is discussed in detail in Section 5. It 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 DFA-based XSD. 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.

3.7 BonXai at Work

The GUI aids to understand the correspondence between BonXai rules and the generated complex types in the transformed XML Schema document. Although the simplicity of BonXai relies on the absence of complex types, *type names* (which can be provided in BonXai annotations, see Section 3) 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. However, in the translation to XML Schema, we use type names (if present) to generate names for XML Schema complex type definitions.

Advanced functionalities of our GUI facilitate schema development and -debugging. In particular, we support the analysis of the relationships between an XML document, a BonXai schema, and a corresponding XML Schema document as follows:

- Highlighting of XML elements matched by a certain BonXai rule or by an XML Schema complex type.
- Highlighting of the rule/type that matches an element in an XML tree.
- Highlighting the BonXai rule corresponding to an XML Schema 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, i.e., for which the schema allows arbitrary content.

We now discuss a few more specific use cases for BonXai to illustrate that BonXai is not just a "readable syntax for XML Schema" but can also be used to perform some more serious tasks more efficiently.

Fig. 8. The jedit-plugin in action. Here we have three editor panes: Top-Left: XML document from Figure 2, Bottom-Left: BonXai schema from Figure 6, Right: an XML Schema equivalent to the BonXai schema on the left. The `section` element is marked orange, as its content model is not correct. The `bold` element is marked red as it is not allowed to appear at this position in the document (note the spelling error). The turquoise font element has been selected by the user in the XML document. The plugin shows where the element is declared in the BonXai schema and XML Schema respectively by highlighting the element declaration in purple. Furthermore it shows where the content model of the selected element is declared by marking its ancestor path (BonXai) and type declaration (XML Schema) in green.

Developing new Schemas / Using BonXai Stand-Alone. As mentioned before, BonXai is not primarily meant as a replacement for XML Schema, but to a large extent it can be used as such. The system can be used to develop schemas from scratch and to debug them. When the schema is finished, XML documents can be validated natively against the BonXai schema. An alternative validation method is to use our conversion routine to XML Schema and validate against the XML Schema document using a third-party XML Schema validator.

For stand-alone use, BonXai's main strength lies in its succinct and transparent way for defining the *structure* of XML documents. BonXai does not (yet) have a syntax for defining XML Schema simple types. Therefore, simple types always need to be imported from an existing XML Schema document One way to do this is to write a structurally very simple XML Schema document that only defines a set of simple types. This XML Schema document can be imported into the BonXai schema, which can then use the simple types from the XML Schema document and define structural aspects through its grammar.

Evolving from a DTD to an XML Schema document. BonXai can be used to move from DTD to XML Schema rather painlessly while, at the same time, taking advantage of the extra expressiveness. One can automatically convert the given DTD into BonXai, add the desired extra structural features directly in the BonXai schema, and convert the result to XML Schema.

Example 3.2. The BonXai schema in Figure 5 is equivalent to the DTD in Figure 4. By only a few modifications it can be extended to the BonXai schema from Figure 6, which can then be exported to an XML Schema document equivalent to the one in Appendix A.

Schema Evolution. Schema evolution refers to updating a schema to reflect a re-structuring of the underlying data. We distinguish two use cases regarding schema evolution, depending on whether we want to modify an existing XML Schema document or an existing BonXai schema using our system. In the latter case, schema evolution can simply be done by editing the BonXai schema. In the former case, the workflow is roughly the following: Convert the XML Schema document to BonXai; alter the schema by specifying additional constraints or changing some content models; and re-export the schema to XML Schema.

The highlighting features of the system, mapping patterns in BonXai rules to complex types in the generated XML Schema fragment provide the developer with control to inspect the induced changes in the original XML Schema more rapidly and accurately.

Especially the priority system used by BonXai can be very helpful in schema evolution. For example, 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 6, 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.

Doing the equivalent change directly in XML Schema requires to make three complex types for sections below `content`: one for each allowed nesting depth.

Analyzing existing XML Schema documents. Existing XML Schema documents can be converted to BonXai to analyze their structural complexity. Such a BonXai inspection can, 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 can give direct insight into the definition of elements depending on their context. As such, the BonXai translation, converting the machine readable syntax of XML Schema 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. The selection patterns in the left-hand sides of BonXai rules give 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.

Example 3.3. In our running example, the BonXai rules

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

and

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

give immediate insight in the difference between the complex types `TtemplateSection` and `Tsection` from the XML Schema Document in Appendix A. The former specifies the structure of `section`-descendants of `template` elements in the tree; and the latter of `section`-descendants of `content` elements.

4 A PRACTICAL STUDY ON THE USE OF COMPLEX TYPES

As a part of the motivation for BonXai, we conducted a large-scale practical study to investigate the extent to which XML Schema complex types are used in practice. A similar study was already conducted over a decade ago by Bex et al. [2] but we decided that we needed more up-to-date information. Whereas Bex et al. [2] studied a corpus of 225 unique SDs, we were able to collect 8080 unique, well-formed SDs from the Web. Our data set was obtained in three steps. We started with the set of 1191 SDs from the practical study in [5] and augmented it with a second set which we harvested using Google's CSE in 2016. We obtained this set by querying Google's CSE for files of type 'xsd', iteratively over all domains listed in Wikipedia[10] and downloading the results. We iterated through date ranges to be able to go beyond the 100 result restriction of Google's CSE. (In particular, whenever we noticed that CSE returned 100 results, we decreased the date range to make sure that we got all results that CSE could give us.) Since some of the results (about 10%; similar as in [5]) are not actual XSD files but HTML documents that link to an XSD file, we extracted these links and downloaded the resulting XML Schema documents. We then compared the results we obtained from Google CSE with Google's Web interface and noticed that the Web interface gives significantly more results (possibly due to the use of the date range operator). We therefore composed a third data set semi-manually. With the help of a student research assistant we manually queried Google's Web interface (again, over every top-level domain mentioned before) and saved the returned links to file. We downloaded these SDs automatically and, again, we considered the HTML files and downloaded the linked XSD files.

The resulting data was cleaned, filtered using an XML Schema parser, normalized for whitespace, and then deduplicated. This resulted in the 8080 unique files that we could parse as XML Schema. We analysed to which extent these SDs use ancestor information to determine complex types (Table 2). We observed the following: 80.66% of SDs associate at most one complex type to every element name. In other words, these SDs do not use complex types beyond the power of DTDs. For a further 13.03%, the complex types were determined by the parent context (as in the example before, with `preface` and `chapter`). A further 3.64% also took the grandparent context into account. The longest finite dependency we found was up to ancestors of height fifteen.[11] As indicated by the entry ∞, we found 35 schemas for which ancestors up to arbitrary height determined the complex

[10] https://en.wikipedia.org/wiki/List_of_Internet_top-level_domains

[11] This particular schema was quite large and validated trees that encode strategies for tic-tac-toe (using two levels in the tree for encoding one move). Schema: https://xopus.com/files/tictactoe/tictactoe.xsd

Table 2. The amount of context information needed to determine complex types in our data set.

k	# schemas	relative	k	# schemas	relative	k	# schemas	relative
1	6517	80.66 %	5	24	0.30 %	9	2	0.02 %
2	1053	13.03 %	6	9	0.11 %	14	1	0.01%
3	294	3.64 %	7	3	0.04 %	15	1	0.01%
4	132	1.63 %	8	4	0.05 %	∞	35	0.43 %
						TA	5	0.06 %

type of an element (which can only happen through recursion in the schema). Finally, the entry "TA" denotes the number of schemas in our corpus that use XML Schema 1.1 *type alternatives*.[12]

To conclude, this practical study shows that the expressiveness of XML Schema complex types is only used very sparingly in practice. Furthermore, it shows that, for the overwhelming majority of Schema Documents, the structure of their complex types can be expressed using BonXai schemas with very simple ancestor patterns. We discuss this matter from a theoretical perspective in Section 5.4.

We made the links to all the .xsd files we found available for download [24]. The data set contains raw links to 79.642 SDs. We also provided a smaller data set of links to unique SDs that were still reachable in May 2017.

5 FOUNDATIONS OF BONXAI

In this section, we define clean formal models for the core of BonXai and XML Schema to explain the principles of our translation from BonXai to XML Schema and back. However, we do not describe the translation of *all* features (such as namespace support, attributes, groups, key/keyrefs, references to types in foreign namespaces, etc.), since this is beyond the scope of this article. Instead, we focus on considerably restricted, clean, cores of BonXai and XML Schema and formally define how we translate these back and forth. Our abstractions will be on the level of tree languages, i.e., the abstraction level of [26, 29]. These abstractions allow us to explain the theoretically most challenging part of the translations that we implemented and to analyse size tradeoffs between the languages.

The theoretical novelty in this section is that the translations that we present here are the first that can preserve XML Schema's expressive power on the level of tree languages. Although other characterizations and translations exist (see [17, 20, 26]), the translations we present here are the first that take into account XML Schema's Unique Particle Attribution (UPA) constraint [16, Section 3.8.6.4]. Indeed, the abstractions in [17, 20, 26] all assume that XML Schema content models correspond to regular languages. However, as noted in Section 3.3, the UPA constraint in XML Schema restricts content model definitions to a *strict subclass* of the regular languages that is *not closed under union or intersection*. This has an important consequence, namely that the *universal semantics* and *existential semantics* of BonXai-like schemas that were studied in [17, 20] are not well compatible with XML Schema (see Section 3.3 for a deeper discussion). Here, we build further on the ideas in [17, 20] to define a *priority-based semantics* for BonXai. We prove that this system does allow for back-and-forth translations that take XML Schema's weaker content model definitions into account and, to the best of our knowledge, is the first to do so.

In summary, we will present the following:

- compact and clear formal models of the core of BonXai and the core of XML Schema, stripped of features that are not essential for analysing the conversion algorithms;

[12] Using type alternatives, the complex type of an element may also be determined by attribute values of its ancestors.

- formal back and forth translation procedures between core XML Schema and core BonXai;
- an analysis of the blow-up of these conversions;
- a proof of worst-case optimality for the conversions; and
- practically relevant fragments of XML Schema and BonXai, where the conversions are efficient.

As a consequence, we obtain that BonXai and XML Schema are equally expressive on the level of tree languages.

5.1 The Theory Underlying BonXai: Core XML Schema and Core BonXai

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

5.1.1 Basic Terminology. We introduce XML trees, regular expressions, and finite automata.

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 $\mathrm{lab}(v) \in \mathsf{EName}$. 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 $\mathrm{lab}(v_1) \cdots \mathrm{lab}(v_n)$, 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 $\text{ch-str}^D(v) = \mathrm{lab}(u_1) \cdots \mathrm{lab}(u_m)$. We note that $\text{ch-str}^D(v)$ is sometimes also called the *content* of node v. We often omit D in the notation of ancestor- or child-strings when it is clear from the context.

Example 5.1. Consider the `section` child v of the element `template` in the tree of Figure 3. 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 for the set $\delta^*(q_0, w)$ of states that A can reach after reading w, starting from its initial state.

We use regular expressions r with the following syntax[13]

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

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

[13] We note that BonXai and XML Schema content models have the same limited support for the `all`-operator (denoted & in BonXai). Similarly, BonXai has the same support of counters (`minOccurs`/`maxOccurs`) than XML Schema. We exclude both here for simplicity, because their back-and-forth translation is straightforward.

expression r to be its total number of alphabet symbol occurrences. For example, both expressions aaa and $a(b+c)?$ have size three.

The Unique Particle Attribution (UPA) of XML Schema specifies that regular expressions in content models need to be *deterministic* [16, Section 3.8.6.4]. We note that such expressions are sometimes also called *one-unambiguous* [6]. Intuitively, a regular expression is deterministic if, without looking ahead in the input string, it allows to match each symbol of that string uniquely against a position in the expression when processing the input in one pass from left to right. For instance, $(a+b)^*a$ is not deterministic as already the first symbol in the string aaa could be matched by either the first or the second a in the expression. Without lookahead, it is impossible to know which one to choose. The equivalent expression $b^*a(b^*a)^*$, on the other hand, is deterministic. Formally, let $\bar{r}$ stand for the regular expression obtained from r by replacing the i-th occurrence of alphabet symbol a in r by a_i, for every i and a. For example, for $r = b^*a(b^*a)^*$, we have $\bar{r} = b_1^*a_1(b_2^*a_2)^*$.

Definition 1 ([6], Definition 2.1). *A regular expression r is* deterministic *(also:* one-unambiguous*) if there are no strings wa_iv and wa_jv' in $L(\bar{r})$ such that $i \neq j$.* ■

Equivalently, an expression is deterministic if the Glushkov construction translates it into a deterministic finite automaton [6]. As a matter of fact, not every regular expression is equivalent to a deterministic one [6]. Thus, semantically, the class of deterministic regular expressions forms a strict subclass of the class of all regular expressions. We note that deciding if for a given regular expression there exists an equivalent deterministic regular expression is PSPACE-complete [11].

5.1.2 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 [23] 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 their translation from and to XML Schema is straightforward.

Definition 2. 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 and s_i are regular expressions over EName, and
- all s_i are deterministic.

■

We call i the *index* of rule $r_i \to s_i$, for each $i = 1, \ldots, n$. Furthermore, we call r_i the *ancestor pattern* and s_i the *child pattern* of the rule. 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.

Our abstraction of BonXai Schema Definitions requires expressions s_i to be deterministic to make BXSDs expressively equivalent to XML Schema's core.

Example 5.2. The formal abstraction of the BonXai schema in Figure 6 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):

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

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

Given the lacking closure of DRE under Boolean operations noted in Section 3.3, it is crucial that none of our conversion algorithms presented in Section 5.2 construct unions, intersections, or complements of content models. In fact, all our conversion algorithms only *copy* these expressions. Therefore, if one were interested in converting between BonXai and a dialect of XML Schema that does not require expressions to be deterministic, one can simply remove the corresponding requirement from BXSDs and use the same conversion algorithms.

5.1.3 A Formal Model for Core XML Schema. Our abstraction of an XML Schema closely follows the definition from [25, 26, 29].

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 3. 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.

■

Throughout this section we consistently use the abbreviation XSD to denote XSchema definitions. We sometimes refer to $\rho(t)$ as the *content model associated to* t. The EDC constraint can be found in [16, Section 3.8.6.3] and, as mentioned before, the UPA constraint in [16, 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 $u \in \text{Nodes}(D)$, we have $\mu(u) \in \{\text{lab}(u)[t] \mid t \in \mathsf{Types}\}$.
- For each node $u \in \text{Nodes}(D)$ with children $u_1, \ldots, u_n$ from left to right, we have $\mu(u_1) \cdots \mu(u_n) \in L(\mu(u))$.

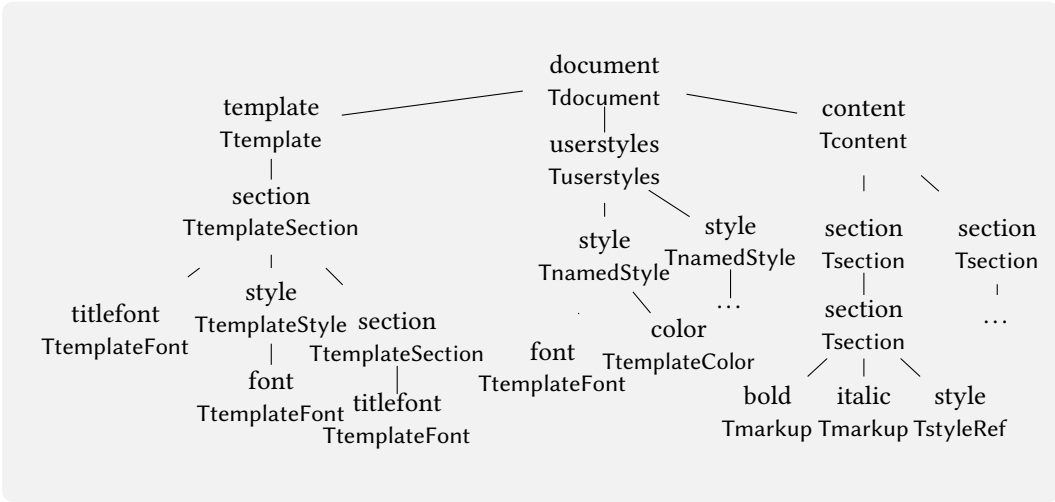

Fig. 9. Typing for the XML document in Figure 2 and the XML Schema Definition in Example 5.3.

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.

Example 5.3. We present a simplified XSchema Definition for our example markup language from Section 2 to illustrate XSchemas. We focus on elements, since this is the part where the complexity lies when converting between XML Schema and BonXai. We can abstract the schema as XSD $X = (\text{EName}, \text{Types}, \rho, T_0)$, where

- EName = {document, content, section, style, bold, italic, font, color, template, userstyles}
- Types = {Tdocument, Ttemplate, Tuserstyles, Tcontent, TtemplateSection, TtemplateStyle, TtemplateFont, TtemplateColor, TnamedStyle, Tsection, Tmarkup, TstyleRef, Tfont, Tcolor}
- ρ is defined as follows (some parts omitted):

  Tdocument → template[Ttemplate] userstyles[Tuserstyles] content[Tcontent]
  Ttemplate → (section[TtemplateSection])?
  Tuserstyles → (style[TnamedStyle])*
  Tcontent → (section[Tsection])*
  TtemplateSection → titlefont[TtemplateFont]? style[Ttemplatestyle]? section[TtemplateSection]?
  Tsection → (bold[Tmarkup] + ⋯ + color[Tcolor] + section[Tsection])*
  ⋯
- T_0 = {Tdocument}

For the sake of the presentation we simplified the example a bit. In particular, we did not specify the function ρ for all types and we omitted rules that would use the xs:all operator (respectively, the &-operator in BonXai).

The correct typing for the XML document in Figure 2 according to the XSD in Example 5.3 is displayed in Figure 9.

5.2 Translations Between Schemas

5.2.1 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 [23]. 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 [25, Definition 6] as an alternative characterization of XML Schema Definitions. We define DFA-based XSDs here with a minor difference: due to the UPA condition, we require their content models to be deterministic regular expressions.

Definition 4. A *DFA-based XSD* 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, $S \subseteq \mathsf{EName}$ 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, q_0 has no incoming transitions and 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 article, 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.

An XML document D *satisfies* (A, S, λ) if the root node is labeled 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 [17, 26], which were developed for different models of XSDs.[14] It has the following property.

Lemma 5 (Adapted from [17, Lemma 7]). *Each XSD can be translated into an equivalent DFA-based XSD in linear time.*

Proof. Let $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ be an arbitrary XSD. The equivalent DFA-based XSD (A, λ) with $A = (\mathsf{EName}, Q, \delta, q_0)$ is constructed by Algorithm 1. We provide additional explanation for the algorithm. In line 3, $\delta(q_0, a)$ is well-defined thanks to the EDC constraint for XSDs (that states that t is uniquely determined by a). Similarly, in line 4 we have that X fulfills the EDC constraint. Therefore, $\delta(t_1, a)$ is well-defined and A is guaranteed to be a deterministic automaton. Finally, in line 5, $\mu(\rho(t))$ denotes the regular expression obtained from $\rho(t)$ by replacing every typed element $a[t]$ by the element a. Notice that, since X fulfills the UPA constraint, we have that $\mu(\rho(t))$ is a deterministic regular expression. Therefore, (A, S, λ) is a DFA-based XSD and has deterministic content models. The fact that (A, S, λ) can be constructed from X in linear time is immediate from the algorithm. The equivalence between (A, S, λ) and X is easily seen. □

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 [26].

Lemma 6. *Each DFA-based XSD (A, S, λ) can be translated into an equivalent BXSD B with linearly many rules in $|A|$.*

[14] One consequence of the slightly different models of XSDs is that the translation in [17] 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 $\{q_1, \ldots, q_n\} = Q$

Proof. Let (A, S, λ) be a DFA-based XSD with $A = (\mathsf{EName}, Q, \delta, q_0)$. Algorithm 2 specifies how to obtain the equivalent BXSD $B = (\mathsf{EName}, S, R)$. In line 2, the regular expression r_q defines the language of the DFA A in which q is an accepting state, i.e., the language of the automaton $(\mathsf{EName}, Q, \delta, q_0, \{q\})$. Since each expression s_q on line 3 is deterministic, the right-hand sides of rules in R are deterministic as well. Finally, R contains the rules $r_q \to s_q$, for each $q \in Q$, in arbitrary order. □

Notice that the ordering of the rules in R in Algorithm 2 can be arbitrary, since, because A is a DFA, $L(r_{q_1}) \cap L(r_{q_2}) = \emptyset$ for each pair of states $q_1 \neq q_2$ from A. We emphasize that the BXSD B can have regular expressions that are exponentially larger than $|A|$ in general. This cannot be avoided[15] because A is a DFA and the worst-case conversion from a DFA to a regular expression is well-known to be exponential [14]. In Section 5.4 we discuss classes of schemas that capture most cases in practice and that do not lead to such a blow-up.

5.2.2 From BonXai to XML Schema. The translation from BonXai to XML Schema follows a similar overall outline as the reverse translation of Section 5.2.1. 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 7. *Each BXSD B can be translated into an equivalent DFA-based XSD (A, S, λ) for which $|A|$ is at most exponential in $|B|$.*

Proof. Let $B = (\mathsf{EName}, S, R)$ be a BXSD, where $R = r_1 \to s_1, \ldots, r_n \to s_n$. We translate B into (A, S, λ) as described in Algorithm 3. On line 2 we want the DFAs $A_i = (\mathsf{EName}, Q_i, \delta_i, q_0^i, F_i)$ to be *minimal* and *complete*. Here, a DFA A_i is *complete* when $\delta_i(q, a)$ is defined for every $q \in Q_i$ and $a \in \mathsf{EName}$. A DFA can be made complete by adding an extra "sink state" to which all previously

[15] Proving that an exponential blow-up cannot be avoided is more technical than just this observation, see Section 5.3.

non-defined transitions lead. Furthermore, it is well-known that every regular language has a unique minimal, complete DFA. (Notice that, since regular expressions are exponentially more succinct than deterministic finite automata, A_i can be exponentially larger than r_i in the worst case.)

The DFA-based XSD (A, S, λ) is then constructed through a product automaton: in line 3, we define A to be the product $A_1 \times \cdots \times A_n$. More precisely, $A = (Q, \mathsf{EName}, \delta, q_0)$, where $Q = Q_1 \times \cdots \times Q_n$, $q_0 = (q_0^1, \ldots, q_0^n)$ and, for every state $(p_1, \ldots, p_n) \in Q$ and every $a \in \mathsf{EName}$, we have $\delta((p_1, \ldots, p_n), a) = (q_1, \ldots, q_n)$ where, for every i, $\delta(p_i, a) = q_i$. Notice that A can be exponentially larger than $|B|$ and does not have accepting states.

The content models of the DFA-based XSD are defined in lines 7 and 9. Line 7 handles the case where at least one of the automata $A_1, \ldots, A_n$ accepts, i.e., at least one BXSD rule matches. The content model of the relevant state in the DFA-based XSD is then defined to be the content of the highest-priority matching BXSD rule. Line 9 handles the case where no BXSD rule matches. Here, according to the definition of BXSDs, every child-string should be allowed. We therefore must allow the content $(\mathsf{EName})^*$. It can be shown that B is equivalent to (A, S, λ). □

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 satisfying 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 8 (Adapted from [17, Lemma 7]). *Each DFA-based XSD can be translated into an equivalent XSD in linear time.*

Proof. Let (A, S, λ) be a DFA-based XSD, where $A = (\mathsf{EName}, Q, q_0, \delta)$. We construct an equivalent XSD $X = (\mathsf{EName}, \mathsf{Types}, \rho, T_0)$ in Algorithm 4. In line 4 of the algorithm we implicitly use that $\delta(q, a)$ is non-empty for every state q and every element name a occurring in $\lambda(q)$. □

We note that the XSD that results from Algorithm 4 can be "minimized" efficiently using a minor adaptation of the minimization algorithm for XSDs from [27].[16] The difference with the minimization algorithm from [27] would be that the deterministic regular expressions r_q should not be minimized.[17]

5.3 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 5.4, we exhibit fragments that are prevalent in practice for which the conversions are efficient.

5.3.1 From XML Schema to BonXai. When converting an XSchema Definition (XSD) to a BonXai Schema Definition (BXSD) using the procedures in Lemmas 5 and 6 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 6. More precisely, line 1 constructs a regular expression equivalent to a DFA, which is well known to be exponential in the worst case [14].

[16] 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.

[17] In fact, it is not clear how to efficiently minimize a deterministic regular expression — if it were possible to do this efficiently, the whole resulting XSD could be minimized in polynomial time by the algorithm from [27].

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: $\rho(q) = r_q$

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. We recall, however, that our conversion which we showed in Lemma 6 does not produce a large number of rules in the BXSD. Therefore, if the DFAs that Algorithm 2 encounters on line 2 only yield polynomially large regular expressions, then the whole conversion is polynomial. We discuss a particularly relevant such case in Section 5.4.

The following theorem is the most technical result in this article. Its proof leverages a technique from [14]. 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 9. *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 $2^{\Omega(n)}$.*

Before we give the proof of Theorem 9, we need a lemma that bounds the size of regular expressions for left derivatives of languages (left derivatives were defined by Brzozowski [7]). To this end, the *left derivative of a string language L with respect to a string w*, denoted by $\partial_w L$, is defined as

$$\partial_w L \stackrel{\text{def}}{=} \{v \mid wv \in L\}.$$

The *left derivative of a language L with respect to a language X*, denoted by $\partial_X L$, is defined as

$$\partial_X L \stackrel{\text{def}}{=} \bigcup_{w \in X} \partial_w L = \{v \mid \exists w \in X \text{ such that } wv \in L\}.$$

For a regular expression α, we denote by $\partial_X \alpha$ a regular expression for the language $\partial_X L(\alpha)$. We denote by $\text{depth}(\alpha)$ the depth of the parse tree for α.

Lemma 10. *Let α be a regular expression and X be an arbitrary language. Then there exists a regular expression α' for the language $\partial_X L(\alpha)$, such that $|\alpha'| \in O(\text{depth}(\alpha)|\alpha|)$.*

Proof. If $X = \emptyset$ then $\partial_X L(\alpha) = \emptyset$ and the lemma holds. We therefore assume from now on that $X \neq \emptyset$. For a language L, let $\text{prefix}(L) = \{v \mid \exists w \text{ such that } vw \in L\}$ be the set of all prefixes of strings in L. We construct α' inductively as follows.

$$
\begin{aligned}
\partial_X \emptyset &= \emptyset \\
\partial_X \varepsilon &= \begin{cases} \varepsilon & \text{if } \varepsilon \in X \\ \emptyset & \text{otherwise} \end{cases} \\
\partial_X a &= \begin{cases} \varepsilon + a & \text{if } X \cap \{\varepsilon, a\} = \{\varepsilon, a\} \\ a & \text{if } X \cap \{\varepsilon, a\} = \{\varepsilon\} \\ \varepsilon & \text{if } X \cap \{\varepsilon, a\} = \{a\} \\ \emptyset & \text{otherwise} \end{cases} \\
\partial_X (\alpha_1 + \alpha_2) &= \partial_X \alpha_1 + \partial_X \alpha_2 \\
\partial_X (\alpha_1 \cdot \alpha_2) &= (\partial_{X_1} \alpha_1) \cdot \alpha_2 + \partial_{\partial_{L(\alpha_1)} X} \alpha_2 \\
\partial_X \alpha^* &= (\partial_{\partial_{L(\alpha^*)} X} (\varepsilon + \alpha)) \cdot \alpha^*
\end{aligned}
$$

Here, $X_1 \stackrel{\text{def}}{=} X \cap \text{prefix}(L(\alpha_1))$ and, by definition $\partial_{L(\alpha_1)} X = \{v \mid \exists w \in L(\alpha_1) \text{ such that } wv \in X\}$ and, similarly, $\partial_{L(\alpha^*)} X = \{v \mid \exists w \in L(\alpha^*) \text{ such that } wv \in X\}$.
We sketch how it can be inductively shown that the above definitions are correct in the case where $X \neq \emptyset$. The base cases are clear. In the inductive step, the set $\partial_X (\alpha_1 + \alpha_2)$ consists of all strings v for which that there is a string $w \in X$ such that $wv \in L(\alpha_1)$ or $wv \in L(\alpha_2)$. This language is defined by the (inductively obtained) regular expression $\partial_X \alpha_1 + \partial_X \alpha_2$. The language $\partial_X (\alpha_1 \cdot \alpha_2)$ is the set of strings v for which there exists a $w \in X$ such that $wv \in L(\alpha_1 \cdot \alpha_2)$. Here, we have two cases depending on how wv matches $\alpha_1 \cdot \alpha_2$. In the first we have $wv = wv_1v_2$ with $wv_1 \in L(\alpha_1)$ and $v_2 \in L(\alpha_2)$ and in the second we have $wv = w_1w_2v$ with $w_1 \in L(\alpha_1)$ and $w_2v \in L(\alpha_2)$. In the first case, we have that w is a prefix of a word in $L(\alpha_1)$ and, therefore, $v \in (\partial_{X_1} \alpha_1) \cdot \alpha_2$. In the second case, $w_2 \in \partial_{L(\alpha_1)} X$ and, therefore, $v \in \partial_{\partial_{L(\alpha_1)} X} \alpha_2$. This concludes the proof for the language $\partial_X (\alpha_1 \cdot \alpha_2)$. We now move on to the final case in the definition, $\partial_X \alpha^*$. Here we must find a regular expression for the strings v for which there is a $w \in X$ such that $wv \in L(\alpha^*)$. Since $wv \in L(\alpha^*)$ we have that wv can be written as $wv = w_1w_2v_1v_2$ where $w = w_1w_2$, $v = v_1v_2$, $w_1 \in L(\alpha^*)$, $w_2v_1 \in L(\varepsilon + \alpha)$, and $v_2 \in L(\alpha^*)$. We have that $w_2 \in \partial_{L(\alpha^*)} X$ and, therefore, v_1 is in the language of $(\partial_{\partial_{L(\alpha^*)} X} (\alpha + \varepsilon))$. This means that v is in the language of $(\partial_{\partial_{L(\alpha^*)} X} (\alpha + \varepsilon)) \cdot \alpha^*$.

It remains to show that $|\alpha'| \leq \text{depth}(\alpha)|\alpha|$. We emphasize that the bound on the length of $\partial_X L(\alpha)$ does not depend on X (and therefore the complicated subscript languages do not matter). This is because the only case where a subscript has a real effect on the expression is for expressions of the form $\partial_X a$.

We show $|\alpha'| \leq 2(\text{depth}(\alpha)|\alpha|)$ by an induction on the structure of α. For the induction base case, we observe that $|\alpha| = |\alpha'| \leq 2$ in the cases where $|\alpha|$ is an atomic expression. Applying the induction hypothesis to the equations above gives us

$$
|\alpha'| \leq \begin{cases} 2\Big(\text{depth}(\alpha_1)|\alpha_1| + \text{depth}(\alpha_2)|\alpha_2|\Big) & \text{if } \alpha = \alpha_1 + \alpha_2 \\ 2\,\text{depth}(\alpha_1)|\alpha_1| + |\alpha_2| + 2\,\text{depth}(\alpha_2)|\alpha_2| & \text{if } \alpha = \alpha_1 \cdot \alpha_2 \\ 2\,\text{depth}(\alpha_1)|\alpha_1| + |\alpha| & \text{if } \alpha = \alpha_1^* \end{cases}
$$

Using the fact that both $\text{depth}(\alpha_1)$ and $\text{depth}(\alpha_2)$ are bounded by $\text{depth}(\alpha) - 1$, we get that

$$
|\alpha'| \leq \begin{cases} 2(\text{depth}(\alpha) - 1)(|\alpha_1| + |\alpha_2|) & \text{if } \alpha = \alpha_1 + \alpha_2 \\ 2(\text{depth}(\alpha) - 1)(|\alpha_1| + |\alpha_2|) + |\alpha_2| & \text{if } \alpha = \alpha_1 \cdot \alpha_2 \\ 2(\text{depth}(\alpha) - 1)|\alpha_1| + |\alpha| & \text{if } \alpha = \alpha_1^* \end{cases}
$$

Using the fact that $|\alpha_1| + |\alpha_2| \leq |\alpha|$, we can conclude in all three cases that $|\alpha'| \leq 2(\text{depth}(\alpha)|\alpha|)$. This concludes the proof. □

Now we are ready to prove Theorem 9: 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 $2^{\Omega(n)}$.

OF THEOREM 9. We leverage a technique by Ehrenfeucht and Zeiger [14], who showed that there exists 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 $O(n^2)$. To show that every BXSD B equivalent to (A_n, S_n, λ_n) is of size $2^{\Omega(n)}$ we prove that B must have at least one ancestor pattern of size $2^{\Omega(n)}$. As already mentioned, it is known from [14] that every regular expression for Z_n is of size $2^{\Omega(n)}$. Actually Ehrenfeucht and Zeiger prove a stronger result:

PROPOSITION 11 ([14, THEOREM 4.1]). *For every $n \in \mathbb{N}$, there is a string $g \in Z_n$, such that every regular expression α with $vgw \in L(\alpha)$ for some v and w and $L(\alpha) \subseteq Z_n$ is of size $2^{\Omega(n)}$.*

For our purposes, we need a slightly stronger version:

PROPOSITION 12. *For every $n \in \mathbb{N}$, there are strings $g_1, \ldots, g_n \in Z_n$ such that $h = g_1 g_2 \ldots g_n \in Z_n$ and for every $i \in \{1, \ldots, n\}$,*

- *g_i contains no symbol from $\{a_{1i}, \ldots, a_{ni}\}$; and*
- *every regular expression α_i with $vg_i w \in L(\alpha_i)$ for some v and w and $L(\alpha_i) \subseteq Z_n$ is of size $2^{\Omega(n)}$.*

of Proposition 12. First we note that Proposition 11 still holds, if we replace the condition $L(\alpha) \subseteq Z_n$ by $L(\alpha) \subseteq Z_m$, for any $m > n$. This is because symbols outside Σ_n are useless for strings from Z_n, and therefore any regular expression for Z_n over Σ_m could be translated into an expression of (at most) the same size over Σ_n by replacing every symbol outside Σ_n with $\emptyset$.

By the same kind of reasoning it follows that, for every $i \in \{1, \ldots, n\}$, Proposition 11 also holds with respect to strings in Z_n over $\Sigma_n^{(i)} = \Sigma_n \setminus \{a_{ij}, a_{ji} \mid j \leq n\}$ and expressions over Σ_n. Let thus, for every i, $h_i \in Z_n$ be a string over $\Sigma_n^{(i)}$ such that every regular expression α with $v_i h_i w_i \in L(\alpha)$ for some v_i and w_i and $L(\alpha) \subseteq Z_n$ is of size $2^{\Omega(n)}$. By choosing v_i and w_i as suitable one-letter strings we obtain strings $g_i = v_i h_i w_i$ with the stated properties. This concludes the proof of Proposition 12. □

Let now B be a BXSD for (A_n, S_n, λ_n). Our goal is to show that B has at least one ancestor pattern of size $2^{\Omega(n)}$. We can assume w.l.o.g. that B does not contain any rule with a child pattern allowing content models $a_{ii}a_{ii}$ and $a_{jj}a_{jj}$, for $i \neq j$. To this end, let us assume such a rule α exists and there is a string $z = a_1 \ldots a_m$ matching the left hand side of α such that some document in $L(B)$ contains z as its ancestor path. If no such z exists, α can be deleted from B without changing its language. On the other hand, if such a document exists, α allows the document in which below the z-path two leaves labeled a_{ii} occur and the document in which below the z-path two leaves labeled a_{jj} occur, contradicting the definition of the language of X_n.

We call any rule allowing a content model $a_{ii}a_{ii}$ a t_i-rule and any other rule a t-rule. We emphasize that, as we just showed, a rule can only be a t_i-rule, for *one* index i.

We consider strings (as ancestor paths) from Z_n of the form $s = h^k s'$, with h from Proposition 12, $k \geq 1$ and $s' \in \Sigma_n^*$. Clearly, strings can be matched by several rules, but for each string s, B must have a last rule $r_s : \alpha_s \to \beta_s$ whose left hand side matches s. However, several strings can possibly share the same last rule.

Let, for every such s,

$$\alpha'_s = \partial_{h^{k-1} g_1 \ldots g_j}\ \alpha_s,$$

where $j = 0$ if α_s is a t-rule and $j = i - 1$ if α_s is a t_i-rule. We note that $g_{j+1} \ldots g_n s' \in L(\alpha'_s)$ by construction. By Lemma 10, it follows that $|\alpha'_s| = O(|\alpha_s|^2)$ and therefore $|\alpha_s| = \Omega(\sqrt{|\alpha'_s|})$.

For each string $s = h^k s' \in \Sigma_n^*$ one of the following conditions must hold, for some $\ell \in \{1, \ldots, n\}$.

(1a) $L(\alpha'_s) \subseteq Z_n$.
(1b) $L(\alpha'_s) \nsubseteq Z_n$, r_s is a t_ℓ-rule, and every string in $L(\alpha'_s) \setminus Z_n$ has error index ℓ.
(2a) $L(\alpha'_s) \nsubseteq Z_n$, r_s is a t_ℓ-rule, and there exists a string in $L(\alpha'_s) \setminus Z_n$ with error index $j \neq \ell$.
(2b) $L(\alpha'_s) \nsubseteq Z_n$ and r_s is a t-rule.

Let us assume first that, for some $s = h^k s' \in \Sigma_n^*$, one of the cases (1a) or (1b) holds.

In case (1a), we can conclude from Proposition 12 that α'_s is of size $2^{\Omega(n)}$. Therefore α_s is of size $\sqrt{2^{\Omega(n)}} = 2^{\Omega(n)}$.

In case (1b), we construct a regular expression γ from α'_s by replacing each occurrence of a symbol $a_{i\ell}$ with $i \in \{1, \ldots, n\}$ by $\emptyset$. By construction, γ has the following properties:

- $|\gamma| \leq |\alpha'_s|$;
- $L(\gamma) \subseteq Z_n$, since every string in $L(\alpha'_s) \setminus Z_n$ has a symbol $a_{i\ell}$ for some i; and
- $g_\ell \in L(\gamma)$, as $g_\ell \in L(\alpha'_s)$ and g_ℓ contains no symbol $a_{i\ell}$ by definition.

We can conclude from Proposition 12, that γ and therefore α'_s is of size $2^{\Omega(n)}$. We can conclude again that α_s is of size $2^{\Omega(n)}$, as well.

We can thus assume from now on that, for every $s = h^k s' \in \Sigma_n^*$, one of the cases (2a) or (2b) applies. We are going to show next that this implies that the number of rules in B must be unbounded,

a contradiction from which we can conclude the statement of the theorem. More precisely, we show that for each string of the form $s = h^k s' \in \Sigma_n^*$, there is a string $z = h^{k-1}z' \in \Sigma_n^*$ such that r_z comes *strictly after* r_s in the list of rules of B. Clearly, repeated application of this statement yields a sequence of at least k rules with ascending indexes. As the process can be started with an arbitrary k, we get the desired contradiction.

Let thus $s = h^k s' \in \Sigma_n^*$, for some $k \geq 1$. By our assumption, either condition (2a) or (2b) holds for α'_s.

We first consider the case that r_s is a t_ℓ-rule, for some $\ell \in \{1, \ldots, n\}$ and (2a) holds with some string $w \in L(\alpha'_s) \setminus Z_n$ with error index $j \neq \ell$. Let us assume towards a contradiction that r_s is the last rule (in the order of rules) matching $z = h^{k-1}g_1 \ldots g_{\ell-1}w$. Then the document consisting of a path with label sequence z arriving at some node v with two leaf children labeled by $a_{\ell\ell}$ below v, is valid for B, a contradiction as the error index of z is not ℓ. Therefore, there must be another rule in B *after* r_s whose left hand side matches z and whose right hand side does *not* allow the content model $a_{\ell\ell}a_{\ell\ell}$.

We next consider the remaining case that r_s is a t-rule and (2b) holds. Let $w \in L(\alpha'_s) \setminus Z_n$ with some error index j and let us assume towards a contradiction that r_s is the last rule matching $z = h^{k-1}w$. Then the document consisting of a path with label sequence z arriving at a node v with two leaf children labeled by a_{jj} is not valid for B, a contradiction.

Therefore, again there must be another rule in B *after* r_s whose left hand side matches z and whose right hand side *allows* the content model $a_{jj}a_{jj}$.

Thus, we have shown that for each string of the form $s = h^k s' \in \Sigma_n^*$, there is a string $z = h^{k-1}z' \in \Sigma_n^*$ such that r_z comes *strictly after* r_s in the list of rules of B, and we are done.

This completes the proof that B has size $2^{\Omega(n)}$. □ □

5.3.2 From BonXai to XML Schema. We prove that the translation from BXSDs to XSDs is worst-case optimal.

THEOREM 13. *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.*

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 [27] 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. □

5.4 Efficient Translations for Fragments

Even though the translations between XSD and BonXai in Sections 5.2.1 and 5.2.2 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. Our examination of 8080 XML Schemas from the Web revealed that, in more than 97%, 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 (see Table 2 in the Appendix; these are the schemas with $k \leq 3$). This data motivates a class of schemas which can be converted efficiently. More precisely, consider the following class of DFA-based XSDs.

Definition 14. 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, 97% of the schemas in our data set have a corresponding 3-suffix DFA-based XSD. Actually, this DFA-based XSD can be obtained simply by applying the construction of Lemma 5 to the given XSD. Furthermore, according to Lemmas 5 and 8, 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 15. 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 [20], 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 16. *Each k-suffix based BXSD can be translated in polynomial time into an equivalent k-suffix DFA-based XSD of linear size.*

Proof. Let $B = (\mathsf{EName}, S, R)$ be a k-suffix based BXSD where $R = (r_1 \to s_1, \ldots, r_n \to s_n)$ where, for each $i = 1, \ldots, n$, $r_i = w_i$ or $r_i = //w_i$ with w_i a string of length at most k.

The equivalent k-suffix DFA-based XSD $D = (A, S, \lambda)$ with $A = (Q, \mathsf{EName}, \delta, q_\varepsilon)$ can be defined as follows. Let $P = \{w \mid \exists \text{ string } v \text{ over EName for which } wv \in \{w_1, \ldots, w_n\}\}$ be the set of prefixes of all w_i and let $Q := \{(q_w, j) \mid w \in P, j \in \{0, 1\}\}$ be a set of states representing all prefixes and indicating whether the "current prefix" is still a prefix of the whole word. Then we define

$$\delta((q_w, j), a) = \begin{cases} (q_v, j), \text{if } wa = v, \\ (q_v, 1), \text{otherwise,} \end{cases}$$

where v is the longest suffix of wa in P. Furthermore we let $\lambda((w, 1)) = s_i$, where i is the highest index such that $r_i = //w_i$ and w_i is a suffix of w, and $\lambda((w, 0)) = \ell$, where ℓ is the highest index such that $r_\ell = w_\ell = w$. The construction of D from B is easily seen to be polynomial. Equivalence

between B and A can be immediately seen since A follows the standard approach for pattern matching with automata. Furthermore, D fulfills the k-suffix property by definition. □

We note that A follows a standard approach for pattern matching with automata. The computed DFA-based XSD is equivalent to the one computed by Algorithm 3. It exploits the fact, that the BXSD is suffix based to avoid the (expensive) product construction.

We now consider the reverse direction. An important difference with Theorem 16 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 97% of the schemas occurring in out practical study, we see that $k \leq 3$.

Theorem 17. *Let k be a constant. Each k-suffix DFA-based XSD can be translated in polynomial time into an equivalent k-suffix based BXSD.*

Proof. Let $D = (A, S, \lambda)$ with $A = (Q, \mathsf{EName}, \delta, q_0)$ be a k-suffix DFA-based XSD. The BXSD $B = (\mathsf{EName}, S, R)$, where B consists of the rules

$$//a_1/a_2/\ldots/a_k \to \alpha, \text{ for which } \lambda(A(a_1a_2\ldots a_k)) = \alpha \text{ and}$$
$$/a_1/a_2/\ldots/a_\ell \to \alpha, \text{ for which } \ell < k \text{ and } \lambda(A(a_1a_2\ldots a_\ell)) = \alpha.$$

Note that the ordering of the rules does not matter as the ancestor patterns describe pairwise disjoint languages.

The BXSD B is equivalent to D and contains less than $|\mathsf{EName}|^{(k+1)}$ rules. It is clear that B can be computed in polynomial time if k is fixed. □

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 [12, 18, 31].

6 CONCLUSIONS

We introduced BonXai with the goal of defining a schema language that approximates the expressiveness of XML Schema and reconciles this with the simplicity of DTDs. Thereby, BonXai is a language in which we can express many features of XML Schema in a more readable format, and automatically convert to actual XML Schema schemas. BonXai is a full-fledged schema language with many features and a formal specification [22]. The language can be employed in various scenarios (c.f., Section 3.7) ranging from the creation of novel XML Schemas to debugging of existing XML Schemas. Furthermore, BonXai is built on a solid theoretical foundation which is rooted in pattern-based schemas [25, 26] and which facilitates transformation algorithms and their analysis. While transforming between BonXai and XML Schema can have high complexity in the worst case, we believe that for a very large and practically relevant class this is never the case (c.f., Section 5.4).

ACKNOWLEDGMENT

We are grateful to Thomas Timm for his significant help in performing the practical study that we discussed in the Introduction. We are grateful to Christian Wolf for collecting the data sets (which included a significant amount of manual labour).

REFERENCES

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

[2] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. 2005. Expressiveness of XSDs: from practice to theory, there and back again. In *International Conference on World Wide Web (WWW)*. 712–721.

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

[4] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. 2010. Inference of Concise Regular Expressions and DTDs. *ACM Trans. Database Syst.* 35, 2 (2010), 11:1–11:47.

[5] Henrik Björklund, Wim Martens, and Thomas Timm. 2015. Efficient Incremental Evaluation of Succinct Regular Expressions. In *International on Conference on Information and Knowledge Management (CIKM)*. 1541–1550.

[6] A. Brüggemann-Klein and D. Wood. 1998. One-Unambiguous Regular Languages. *Information and Computation* 142, 2 (1998), 182–206.

[7] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494.

[8] Russel Butek and Shannon Kendrick. 2011. Web services hints and tips: avoid anonymous types. http://www.ibm.com/developerworks/webservices/library/ws-avoid-anonymous-types/ws-avoid-anonymous-types-pdf.pdf. (2011). IBM developerWorks Technical Library.

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

[10] Claudio Sacerdoti Coen, Paolo Marinelli, and Fabio Vitali. 2004. Schemapath, a minimal extension to XML schema for conditional constraints. In *WWW*, Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills (Eds.). ACM, 164–174.

[11] W. Czerwiński, C. David, K. Losemann, and W. Martens. 2013. Deciding Definability by Deterministic Regular Expressions. In *International Conference on Foundations of Software Science and Computation (FOSSACS)*. 289–304.

[12] W. Czerwiński, W. Martens, and T. Masopust. 2013. Efficient Separability of Regular Languages by Subsequences and Suffixes. In *International Colloquium on Automata, Languages, and Programming (ICALP)*. 150–161.

[13] DSD. 2002. Document Structure Description (DSD). http://www.brics.dk/DSD/. (2002).

[14] A. Ehrenfeucht and H. P. Zeiger. 1976. Complexity Measures for Regular Expressions. *J. Comput. Syst. Sci.* 12, 2 (1976), 134–146.

[15] Davide Fiorello, Nicola Gessa, Paolo Marinelli, and Fabio Vitali. 2004. DTD++ 2.0: Adding support for co-constraints. In *Extreme Markup Languages*.

[16] S. Gao, C.M. Sperberg-McQueen, H. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. 2012. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/. (April 2012).

[17] Wouter Gelade and Frank Neven. 2011. Succinctness of pattern-based schema languages for XML. *J. Comput. System Sci.* 77, 3 (2011), 505–519.

[18] P. Hofman and W. Martens. 2015. Separability by Short Subsequences and Subwords. In *International Conference on Database Theory (ICDT)*. 230–246.

[19] JEdit [n. d.]. jEdit Programmer's Text Editor. www.jedit.org. ([n. d.]).

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

[21] K. Losemann, W. Martens, and M. Niewerth. 2012. Descriptional Complexity of Deterministic Regular Expressions. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*. 643–654.

[22] 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. 2015. Design of the BonXai Schema Language. (Draft 2015). Available at http://www.bonxai.org/downloads/bonxai-design.pdf.

[23] W. Martens, F. Neven, M. Niewerth, and T. Schwentick. 2012. Developing and Analyzing XSDs through BonXai. *PVLDB* 5, 12 (2012), 1994–1997.

[24] Wim Martens, Frank Neven, Matthias Niewerth, and Thomas Schwentick. 2017. BonXai: Combining the simplicity of DTD with the expressiveness of XML Schema (Data Set). http://bonxai.org/downloads.html. (2017).

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

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

[27] W. Martens and J. Niehren. 2007. On the minimization of XML Schemas and tree automata for unranked trees. *J. Comput. System Sci.* 73, 4 (2007), 550–583.

[28] Anders Møller and Michael Schwartzbach. 2006. *An introduction to XML and web technologies*. Addison-Wesley.

[29] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. 2005. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.* 5, 4 (2005), 660–704.

[30] D. Peterson, S. Gao, A. Malhotra, C.M. Sperberg-McQueen, H. Thompson, and P.V. Biron. 2012. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/. (April 2012).

[31] T. Place, L. van Rooijen, and M. Zeitoun. 2013. Separating Regular Languages by Piecewise Testable and Unambiguous Languages. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*. 729–740.
[32] RelaxNG. 2001. Relax NG Specification. http://www.relaxng.org/spec-20011203.html. (2001).
[33] Schematron. 1999. Schematron. http://www.schematron.com/. (1999).
[34] C.M. Sperberg-McQueen and H. Thompson. 2005. XML Schema. http://www.w3.org/XML/Schema. (2005).

A COMPLETE XML SCHEMA DOCUMENT

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns="http://example.org"
       xmlns:xs="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified"
       targetNamespace="http://example.org">
 <xs:element name="document" type="Tdocument"/>
 <xs:complexType name="Tdocument">
   <xs:sequence>
     <xs:element name="template" type="Ttemplate"/>
     <xs:element name="userstyles" type="TuserStyles"/>
     <xs:element name="content" type="Tcontent"/>
   </xs:sequence>
 </xs:complexType>
 <xs:complexType name="Ttemplate">
   <xs:sequence>
     <xs:element name="section" type="TtemplateSection" minOccurs="0"/>
   </xs:sequence>
 </xs:complexType>
 <xs:complexType name="TuserStyles>
   <xs:sequence>
     <xs:element name="style" type="TnamedStyle"
                 minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
 </xs:complexType>
 <xs:complexType name="Tcontent">
   <xs:sequence>
     <xs:element name="section" type="Tsection"
                 minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
 </xs:complexType>
 <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="TtemplateStyle">
   <xs:all>
     <xs:element name="font"  type="TtemplateFont"   minOccurs="0"/>
     <xs:element name="color" type="TtemplateColor"  minOccurs="0"/>
   </xs:all>
 </xs:complexType>
```

Fig. 10. An XML Schema document equivalent to the BonXai schema in Figure 6, describing the XML document in Figure 2 — part 1.

```
<xs:complexType name="TtemplateFont">
  <xs:attribute name="name" type="xs:string"  use="optional"/>
  <xs:attribute name="size" type="xs:integer" use="optional"/>
</xs:complexType>
<xs:complexType name="TtemplateColor">
  <xs:attribute name="color" type="xs:string"/>
</xs:complexType>
<xs:complexType name="TnamedStyle">
  <xs:all>
    <xs:element name="font"  type="TtemplateFont"  minOccurs="0"/>
    <xs:element name="color" type="TtemplateColor" minOccurs="0"/>
  </xs:all>
  <xs:attribute name="name" type="xs:string" use="required"/>
</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:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="bold"   type="Tmarkup"/>
    <xs:element name="italic" type="Tmarkup"/>
    <xs:element name="style"  type="TstyleRef"/>
    <xs:element name="font"   type="Tfont"/>
    <xs:element name="color"  type="Tcolor"/>
  </xs:choice>
</xs:group>
<xs:complexType name="Tmarkup" mixed="true">
  <xs:group ref="markup"/>
</xs:complexType>
<xs:complexType name="TstyleRef" mixed="true">
  <xs:group ref="markup"/>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="Tcolor" mixed="true">
  <xs:group ref="markup"/>
  <xs:attribute name="Tcolor" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="Tfont" mixed="true">
  <xs:group ref="markup"/>
  <xs:attribute name="name" type="xs:string"  use="optional"/>
  <xs:attribute name="size" type="xs:integer" use="optional"/>
</xs:complexType>
</xs:schema>
```

Fig. 11. An XML Schema document equivalent to the BonXai schema in Figure 6, describing the XML document in Figure 2 — part 2.