XStruct: Efficient Schema Extraction from Multiple and Large XML Documents

Jan Hegewald, Felix Naumann, Melanie Weis
Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin
{hegewald,naumann,mweis}@informatik.hu-berlin.de

Abstract

XML is the de facto standard format for data exchange on the Web. While it is fairly simple to generate XML data, it is a complex task to design a schema and then guarantee that the generated data is valid according to that schema. As a consequence much XML data does not have a schema or is not accompanied by its schema. In order to gain the benefits of having a schema—efficient querying and storage of XML data, semantic verification, data integration, etc.—this schema must be extracted.

In this paper we present an automatic technique, XStruct, for XML Schema extraction. Based on ideas of [5], XStruct extracts a schema for XML data by applying several heuristics to deduce regular expressions that are 1-unambiguous and describe each element's contents correctly but generalized to a reasonable degree. Our approach features several advantages over known techniques: XStruct scales to very large documents (beyond 1GB) both in time and memory consumption; it is able to extract a general, complete, correct, minimal, and understandable schema for multiple documents; it detects datatypes and attributes. Experiments confirm these features and properties.

1 Schema Extraction from XML Documents

XML is a standard for storing information in a self-structured way, i.e., documents contain the data as well as its structure. However, the structure is included only implicitly and furthermore only the specific part of the structure that is relevant for that special XML document.

To make the general structure explicit, the W3C introduced DTD and XML Schema as possibilities to define a schema independently of a concrete XML document. XML Schema should be preferred over DTD, because it supports datatypes, more flexible keys, foreign keys, and namespaces, to name just a few advantages. Moreover XML Schema itself is written in XML syntax and therefore can be processed using standard XML processors. The existence of a schema to an XML document offers several advantages:

- Today, in many companies XML serves as a format for data exchange. Especially in the field of data exchange between companies, the recipient has normally little influence on the quality of the data. In the case of an automated processing of the XML data this can lead to problems. In this case a schema enables companies to define the format precisely and, maybe even more importantly, to validate data against that specification, allowing the detection of invalid formats and therefore raising the quality of operational data.

- A second key point is the ability of query processors and XML storage systems to answer queries against XML data more efficiently when its schema is known. By employing optimization techniques, such as query pruning and query rewriting, the algorithms may avoid exploring a large part of the search space unnecessarily by exploiting the knowledge about the data's structure.

To make use of these advantages for documents whose schema is not known, it should be extracted from the raw XML data. A schema for XML data is particularly interesting when dealing with multiple XML documents representing similar data. XStruct addresses these two key tasks and is able to construct a schema in a form of an XML Schema file for a collection of XML documents.

When extracting a schema for a document there are some criteria defining the quality of the generated

schema [1]. These are:

- **Correctness** The produced schema must be of a kind, that the input XML document conforms to that schema. Schemata are especially important for a collection of documents, since in that case all of the processed documents must be valid with respect to the extracted schema.
- **Conciseness** The generated schema should be generalized in a way that the schema may cover the entire structure of all documents with a short definition.
- **Precision** On the other hand, we wish to have a schema that is not too general, which means that the schema should not be valid for too many other documents whose structure is quite different from the input ones.
- **Readability** Moreover, the schema should be easily human-readable and optimally close to schemas that humans would have designed.

As one can see there is a trade-off between the second and third criterion, which are both determinant factors for the fulfillment of the fourth requirement. XStruct produces schemas that are of good quality regarding these criteria.

2 Related Work

The algorithms used in XStruct for inferring regular expressions for an XML element's children are based on ideas of Min et al. [5]. XStruct extends their algorithm to process multiple XML documents instead of only one, therefore being able to produce one schema that is valid for many XML documents. Further enhancements are described later.

The factoring algorithm of Garofalakis et al. [4] for their XTRACT schema extraction application is used to merge the inferred regular expressions for all elements of one type into a single regular expression. In contrast to the work presented in [5], XTRACT is able to process a collection of XML documents as its input and to create a schema that is valid for all of the documents. However, this schema may be incorrect since the 1-unambiguity constraint, which is mandatory for DTDs and XML Schema according to their definition by the W3C, is not guaranteed in the generalization algorithm of XTRACT, as Romberg shows in [7]. Therefore the generated schemas are possibly not correct ones. XStruct extends the ideas of both algorithms and enriches the generated output with type and attribute information of the input XML data.

Other work on extracting schemas from semistructured data includes [6], [3], and [8]. In [6], the schema for semi-structured data is in the form of a monadic Datalog program with each intensional predicate defining a separate type. The authors show that a schema has to allow a certain degree of freedom, because a perfect schema's size (which easily gets as big as the data itself) is of no practical use in databases (e.g., for query optimization). Further, the authors present heuristics to efficiently treat the problem of reducing the number of types, which in general is NP-hard. However, their main focus is the quality of the result, and not—as ours—time and memory performance. In [3], the author proposes a schema extraction method based on the powerful model of extended context-free grammars that supports "new XML schema languages" proposed to replace DTD. This approach is the first step towards extracting schemas conforming to the XML Schema standard. But again, efficiency is not an issue. The authors of [8] propose extracting an approximate graph schema based on an incremental clustering method that clusters vertices with similar incoming and outgoing edge patterns. All these approaches have in common that they consider schema extraction for semi-structured data before XML (in the case of [6]) or XML Schema became standards. Hence, they do not consider XML Schema specific issues, such as extracting datatypes and attributes.

3 The XStruct Algorithm

Within the following sections we distinguish two different views of an XML element. On the one hand we speak of elements, which normally denote a tag in an XML file with a special name, e.g., "book". Thereby we refer to the abstract element with its type. On the other hand we speak of occurrences of an element. That is a concrete instance of the element in the XML data. Specifically, one element may have many occurrences in XML data.

XStruct consists of five modules that each fulfill a special task. A high-level overview of the architecture of XStruct is shown in Figure 1.

A collection of XML documents serves as the input for a SAX parser, which processes them consecutively. The decision to use a SAX parser is a key point for XStruct's ability to scale to large and multiple documents with respect to time and memory consumption. In order to apply the algorithms presented in [5], it was necessary to find a possibility to store the data retrieved during parsing, as the authors of [5] use a DOM parser in the sample implementation of their ideas, which makes access to the data much easier. More-

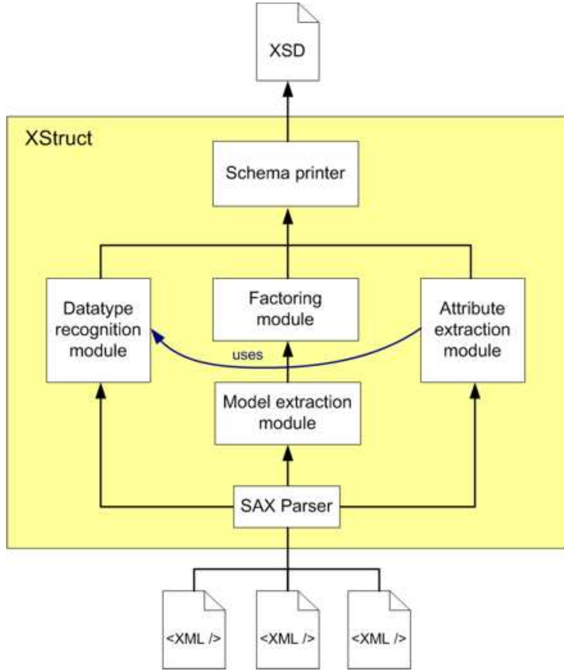

Figure 1. Architecture overview of XStruct

over the format in which the data is stored is another key point regarding XStruct's scalability and therefore we describe our data structures separately in Section 4.

While reading the XML documents the SAX parser passes the data to three modules of XStruct: the model extraction module, the attribute extraction module and the datatype recognition module. The model extraction module afterwards passes its output to the factoring module. We describe each of these modules in the following subsections. Finally the schema printer module takes the output of the factoring, the datatype recognition, and the attribute extraction module and constructs the final XML Schema file.

One advantage of this technique is the ability to extend XStruct to extract a schema iteratively, i.e., it would be possible to parse a number of XML documents, determining their schema and later on parsing some further XML documents, adapting the pre-extracted schema to also suite the newly parsed documents. This feature is not implemented yet, but to add it, it solely would be necessary to store the generated element content models generated in the first session and add the new ones parsed in the second. After each parsing the compliant schema could be output.

3.1 Finding Element Content Models

XML schema specification languages, such as DTD and XML Schema, use an approach for describing the structure of documents by describing the possible child elements of one element. DTDs use a form of regular expressions to do this, XML Schema offers a different syntax, that not only allows to use Kleene-Stars and the "?" for the repetition of elements, but also to specify how often exactly one element may appear with *min* and *max* values. So in order to describe an XML document's structure, we need to describe each element's possible children. Since their children are also described in that way, one is able to specify the entire document structure.

The algorithm of [5] calls the information about one element's children the *element content model*. The authors formally define an element content model in the following way:

$$E := (T_1 \dots T_k)^{<min,max>}$$

where each T_n is a *term*. A term is defined as a set of symbols, either as a sequence or as a choice:

$$T_n := (s_{n1}^{opt} \dots s_{nj}^{opt})^{<min,max>} \qquad \textit{sequence term}$$

or

$$T_n := (s_{n1}^{opt} | \dots | s_{nj}^{opt})^{<min,max>} \qquad \textit{choice term}$$

where $min \in \{0,1\}$, $max \geq 1$, and $opt \in \{true, false\}$.

A symbol s_{lm} is an identifier for an XML element, e.g., the element's name. Thus, a term describes a list of XML elements, that may appear consecutively or alternatively between *min* and *max* times. Each symbol is denoted by *opt*, specifying whether it must appear in the term ($opt = false$), or not ($opt = true$). An element content model as a list of such terms on the other hand specifies in which order those terms must appear and how often the list may be repeated. Thus, an element content model is the representation of a regular expression extended by *min* and *max* values and describes the pattern of possible children of an element. This clearly is a limitation of regular expressions as already Min et al. stated, since for example the regular expression $(ab(c|d*))$ cannot be represented in an element content model: The choice $(c|d)$ is only expressible by a term, which in turn does not support the specification of the number of repetitions for single symbols, as would be necessary for $(c|d*)$. To express such patterns, an element content model can be constructed that is slightly generalized: $E = (T_1T_2)$ with $T_1 = (ab)$ and $T_2 = (c|d)^{<1,\infty>}$.

Min et al. proposed an algorithm that constructs an element content model for every appearance of an XML element. A special constraint of DTD and XML

Schema that has to be taken care of when doing this, is the 1-unambiguity constraint. In general, every regular expression defines a regular language. Simply speaking, the 1-unambiguity constraint requires that for a given word of a given language there may exist only one possible derivation by which the word could have been constructed from the regular grammar. For example, for the grammar defined by the regular expression $a * b?a*$ it is not clear whether the word a, which is a word of the grammar, was derived by the first or the second $a*$ of the regular expression. Therefore, $a * b?a*$ is not an 1-unambiguous grammar. The ISO standard for the Standard Generalized Markup Language SGML enforces that every SGML language description must be 1-unambiguous. Since XML is a subset of SGML it is indispensable that every schema for XML data is also 1-unambiguous. In general, whether a regular expression is 1-unambiguous or not can be determined by creating a Glushkov automaton for that regular expression. If and only if that automaton is deterministic, the regular expression is 1-unambiguous as proven in [2]. Instead, the authors of [5] chose a different approach, which makes use of the fact, that a regular expression that contains every symbol only once, is obviously always 1-unambiguous. Their algorithm consists of nine heuristics plus an additional rule that create an element content model for the children of an XML element. This element content model contains every symbol only once and therefore is 1-unambiguous.

XStruct extends the ideas of [5] in a quite natural way. While [5] originally extracts a schema for only one XML document, the method can be modified to support multiple XML documents as follows: Since the algorithm collects the child elements of every occurrence of an element and creates an element content model for each, which are later factored to one common element content model for that element, one can let the parser scan another XML document as well before the factorization step, so that the number of element content models for each element is increased by the number of occurrences of that element in the second document when ignoring XStruct's ability to omit already found element content models for an element. After parsing the second document, all those element content models for all elements found during parsing can still be factored to one common element content model.

If the second document does not contain some of the first document's elements, this is no problem, since then the resulting element content model is identical to that which would have been constructed when parsing only the first document. It seems intuitive not to change an element content model if the specific XML element does not occur again afterwards. If, on the other hand, the second XML document contains elements that were not present in the first one, then they are added to the list of elements and the element content models is constructed as usual. This decision is also appropriate, because the new element undoubtedly has to be represented in a schema for both documents.

A key point is the handling of each document's root element. Generally, one could treat it like an ordinary element as it is done in the case of a single document. Furthermore, XML Schema also offers the possibility to have different root elements in different documents that comply with the same schema. The reason for this counterintuitive ability is that XML Schema does not specify a root element at all. Every datatype in an XML Schema that is defined on top-level in the schema file can be the root element of an XML document of this schema. This means, that if a schema contains descriptions of several user defined complex types on the top-level each one can serve as a root element and it is impossible to single out one of them as a root. Since we consider it poor modeling practice to use ambiguous root elements, XStruct extracts schemas for only such collections of documents where each document contains the same root element. As a side note, this flexibility of XML Schema is unnecessary, because in the unlikely case that there is good reason to use different root elements, this case could easily be accomplished by introducing an artificial root element that may contain the others. However, because the root element is treated similar to other elements, the only speciality XStruct has to take care of when analyzing multiple documents is to check whether they contain the same root element. From there on, the root of each document is treated like all the other elements. Therefore XStruct infers an element content model that is appropriate for each single XML document as well as the entire collection of them.

The algorithm of [5] and ours cannot detect the ability of XML Schema to have two elements of the same name that have different types, i.e., different regular expressions for their children, when their type can be determined by where the element is located in the document. This is quite complex though, since one needs much information about the document's structure - more than what is stored in XStruct's structure model as described in Section 4. Again, we believe that it is not good modeling style to have elements of different types with the same name. Note anyhow that if documents that contain elements of the same name but with different types are used as input for XStruct, it treats them as of one type and therefore probably over-generalize the element content model.

3.2 Factoring

After the parsing of XML documents, XStruct has created a list of element content models for each element. So after constructing the model the task of integrating all the models of one element into one remains. Formally, all those models could be connected by logical ORs, because every given occurrence of this element conforms to at least one of the models. Unfortunately, we may expect to have many element content models that are similar but not identical. In order to avoid creating long lists of alternatives, which would harm a schemas conciseness and readability, we integrate these models by factoring common prefixes and suffixes.

The algorithm we employ is based on ideas of XTRACT [4], but is adapted to our needs, because XStruct uses different models as input to the factoring module. The theoretical background is from the field of factoring Boolean expressions. As one can treat choices similar to logical ORs and sequences similar to logical ANDs, the element content models are quite similar to Boolean expressions. However, there are some important differences regarding commutativity and *min* and *max* numbers that had to be considered when adapting the algorithm.

Intuitively common prefixes and suffixes get factored, while the remaining parts of a model are expressed as alternatives. Thereby the *min* and *max* values have to be considered. To create valid factoring results, the new *min* and *max* numbers have to be the minimum of all factored *min* numbers and the maximum of all factored *max* numbers, respectively. For example, the models $(1, 2, 3, 4)^{<2,4>}$ and $(1, 2, 5, 4)^{<1,3>}$ when factored result in the following expression in regular expression-like notation: $(1, 2)(3|5)(4)^{<1,4>}$.

3.3 Extracting datatypes

Extraction of primitive datatypes can be a tedious task when considering all 44 built-in primitive and derived datatypes from the XML Schema specification[1]. Therefore, we restrict ourselves to the most commonly used primitive datatypes, that is String, Boolean, Decimal, Integer, Double, Date, and Time. We consider String as the most general type. A definition of the lexical space defining each type can be found in the XML Schema specification. We believe that our choice of datatypes recognized by XStruct covers most real life requirements and note that it can be extended easily.

XStruct's datatype recognition module gleans some extra information for learning datatypes from the content of objects, i.e., of XML elements as well as of attributes. For every object there is a list of user definable capacity that saves all previously saved different contents for that object. At every parsed content of an object, the datatype recognition module is given the information of the previously determined datatype of that object, the list with former contents, and the newly read content. The datatype recognition module returns a new datatype for the object by evaluating this data as follows: Firstly, XStruct determines whether the new content is of the same datatype as the object's previously determined one.

- If the new content is of the type that was determined before, the datatype reported for this object remains the same as before.

- If this is not the case, XStruct tries to find the next suitable type according to the graph shown in Figure 2. This figure is different from the one presented in the XML Schema definition for two reasons: Firstly, XStruct extracts only a subset of the possible datatypes. Secondly, this figure shows the hierarchy of the datatypes' lexical space. Finally, the dotted line in the figure stands for a relation between datatypes, that is not strictly hierarchical, but loose. XStruct considers an object's datatype always as strict as possible, i.e., if one object always seems to be a Decimal, XStruct will treat it as such. If then the object appears with a content that is of type Double, it will be treated as Double, since every Decimal is lexically also a legal Double without exponent. The dotted line shows a relation that is based mainly on lexical familiarity. Every Boolean value may be encoded as one of $\{true, false, 1, 0\}$. Therefore it is possible to have an object always having 1 or 0 as content, making it appear as Boolean for XStruct. When afterwards, a content of 10 is read, according to hierarchy the type had to be considered as String, but XStruct uses the list of formerly read contents to detect that the Boolean value always was encoded as 1 or 0 and not as *true* or *false*, effectively enabling XStruct to discard Boolean as the datatype and decide to choose Integer as the new datatype when reading the content of 10.

Furthermore, XStruct uses the list of formerly read different contents of an object in the schema printer that may, according to a user defined threshold, decide to treat an object with only few different values as an enumeration instead of a primitive type. The list of different contents is moreover used to determine fixed value objects.

[1] http://www.w3.org/TR/xmlschema-2/

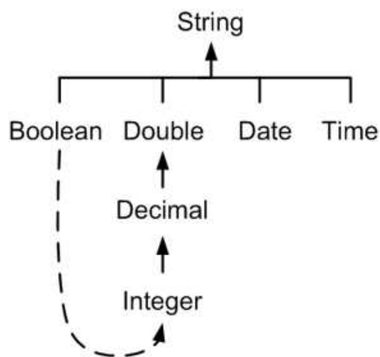

Figure 2. Overview of XStruct's recognized datatypes. Solid lines describe strictly hierarchical relations, the dotted line a loose relation.

Since that list may consume much memory during scanning time, the user is allowed to control the maximal capacity by an option of XStruct. If XStruct scanned more different contents than the maximal capacity of the list, all data in the list is discarded and the object is marked as no longer having a list of former contents in order to prevent erroneous datatype recognition: If, for instance, those values are discarded that prevent a Boolean from being turned into an Integer, an incorrect datatype might be chosen.

We believe that our choice of datatypes recognized by XStruct covers most real life requirements but can be extended easily.

3.4 Extracting Attributes

Extracting an element's set of attributes is a rather straightforward task. According to XML well-formedness requirements, an element may contain an attribute only once, and attributes can be optional or mandatory. The latter can be determined by checking whether this attribute is present in every element of the specific type. During parsing XStruct creates a list of attributes that occurred for every element. If one attribute occurred in all former occurrences of this element, it is considered mandatory. If it is missing in one occurrence, it is regarded as optional. XML Schema's attribute groups are not detected, but this extension is trivial. What XStruct does perform is the detection of fixed value attributes. These are attributes that, if they are present, have to have a predefined value. Detection of default values is not yet implemented and may be part of future work.

To determine the datatype of the recognized attributes, the attribute extraction module uses methods offered by the datatype recognition module. To be able to detect enumeration types, the data structure used to store the attribute information also allows to store a list of values so that the datatype recognition module can be provided with this information as well, which is also necessary to determine fixed values.

4 Data Structures

As mentioned earlier, the employment of a SAX parser makes it necessary to have an efficient data structure that stores data retrieved during scanning. Although it is possible to do the step of creating the element content model of one occurrence of one XML element "on the fly" while parsing, it is impossible to do the factoring immediately, since all occurrences of one element must have been seen before being able to factor one common regular expression for that element. Furthermore it is necessary to store information about the attributes and about the datatypes of both elements and attributes.

To not reinvent a DOM tree, which stores the entire tree representation of XML documents (and effectively destroy XStruct's scalability advantages), XStruct uses a data structure called *structure model*, which contains only the information necessary for the factorization, attribute extraction, and datatype recognition modules. The structure model contains a hash table that stores one entry for every abstract XML element, i.e., not for every occurrence of one element. This entry maps the element's name to an index that gives the position of this element in the second part of the structure model—a list of *XMLElements*, a further data structure.

An XMLElement stores for every type of element in the documents its name, its attributes, its datatype and—most importantly—a list of element content models for that element. An element content model by itself consists of a list of terms and meta data as mentioned earlier.

The structure model is populated as follows: Whenever the parser encounters the opening tag of an element, it adds the element to the element's parent's queue of children, which can be determined since XStruct maintains a stack of currently open parent elements. Additionally the attributes and the datatype are processed and the result is stored in the structure model. When an element's occurrence is closed, the parser activates the model extraction module to transform the queue of child elements into an element content model for that occurrence.

So after the parsing run, the structure model consists of a list of all XML element types that were found

in the documents, each of them containing a list of element content models. XStruct creates the element content model for every occurrence of an element, but it stores only those, that have not already been saved for that element. As one can imagine, if documents are regular, many occurrences of an element are evaluated to the same element content model and therefore this ability of XStruct reduces memory consumption. For an illustrative figure that exemplifies XStruct storing an XML documents structure in its data structure, see Figure 3. All in all, the advantages a structure model

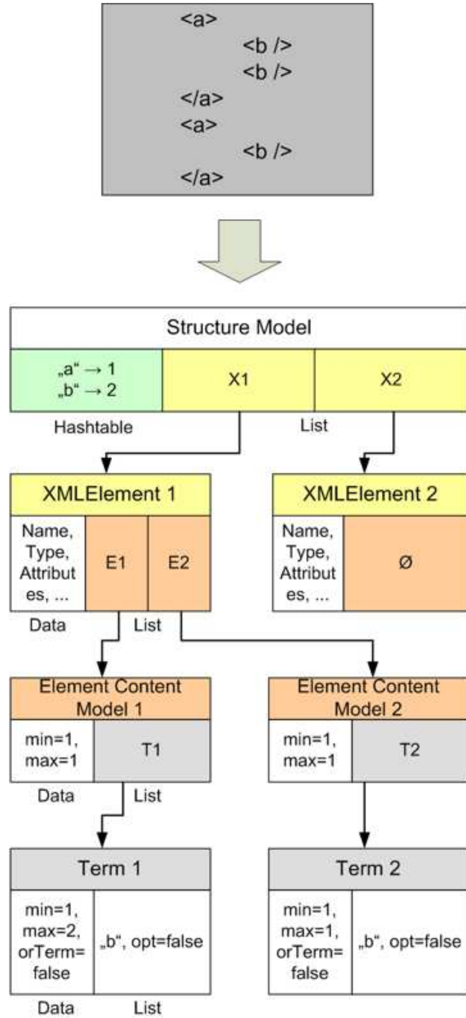

Figure 3. Example of XStruct storing an XML document's structure in its data structures

offers compared to a DOM tree are that it is much more compact in its size: On the one hand information about the exact order and parent-child relationship of elements can be and is indeed ignored, because it is not needed by the factoring module that processes this data. On the other hand only the already compacted element content models are stored (the generalized regular expressions) and furthermore a concrete element content model is stored only once for an element. When scanning large documents with repeating, but structural identical elements, such as thousands of book elements, each of them containing exactly one title and one author, this element content model is represented only once and not a thousand times. For typical applications of XStruct this property is indeed the case and then memory consumption of the algorithm is truly small.

Note that it depends on the structure of the XML file how much memory is needed. If there are many elements whose element content model is equal, XStruct could require only a very small fraction of the memory a DOM would need. If the data is highly irregular and contains very few elements with the same element content model the memory consumption could grow to around the size of the XML data itself (which still is much less than the DOM tree of common DOM implementations requires). Further note that to have the same element content model, two occurrences of one element need not to have exactly the same child elements since the element content model is already a generalized regular expression and slightly different child elements could potentially lead to the same element content model.

5 Experiments

In this section we present results of testing XStruct, evaluating three main aspects: (i) time-scalability with respect to the input document's size (ii) memory consumption with respect to the input document's size, and (iii) differences when parsing many documents versus scanning one document. We do not study the quality of the generated schemas since the utilized algorithms from [5] and [4] were not significantly modified with respect to the output generated by them. As shown in [5] the generated schemata are concise, correct, and complete. We successfully verified that the datatypes and attributes extracted by XStruct are correct, but do not present results here. The XStruct tool and the experimental data are available at `http://www.informatik.hu-berlin.de/mac/xstruct/`.

5.1 Data sets

For stress-testing XStruct's scalability we used XML data from the data generator *xmlgen* of the XMark[2]

[2] `http://monetdb.cwi.nl/xml/`

project—a benchmark for XML database systems and query processors. The data generator produces data similar to real-life e-commerce data. It models the business data of an online auction platform with an XML document of diverse structure. The generated data also includes text content for the elements taken from Shakespeare's plays. xmlgen permits the creation of XML documents of user-defined size while preserving the documents inherent structural complexity.

5.2 Scalability

To verify XStruct's scalability we performed tests with documents of the following sizes: 10 MB, 20 MB, 50 MB, 100 MB, 200 MB, 500 MB, 1 GB (the xmlgen size parameters were approx. set to 0.1, 0.2, 0.5, 1, 2, 5, and 10, respectively). Note that for this series of tests we only used a single XML document as input to XStruct. For results regarding the schema extraction of multiple documents, see Section 5.3.

To measure the amount of memory occupied by XStruct, we used the tool Process Explorer from Sysinternals[3]. Note that the measured values comprise all memory allocated by the process, and therefore also contain memory overhead by the Java virtual machine itself and general program overhead. The time required for extracting the schema is measured by XStruct itself. All experiments were performed on a Pentium 4 M processor at 1.9 GHz with 768 MB main memory on Windows XP and JDK 1.5.

The results confirm that XStruct's data structures impressively minimizes the amount of necessary main memory to extract the schema. See Table 1 and Figure 4 for the detailed results.

Input size	Time required	Memory allocated
11 MB	15 s	24 MB
23 MB	43 s	32 MB
56 MB	109 s	41 MB
113 MB	199 s	51 MB
227 MB	328 s	60 MB
568 MB	559 s	74 MB
1,137 MB	861 s	103 MB

Table 1. Results of the scalability experiments.

Even without loading data, XStruct's memory consumption starts at around 20 MB. This overhead is a result of our measurement: Process Explorer reports

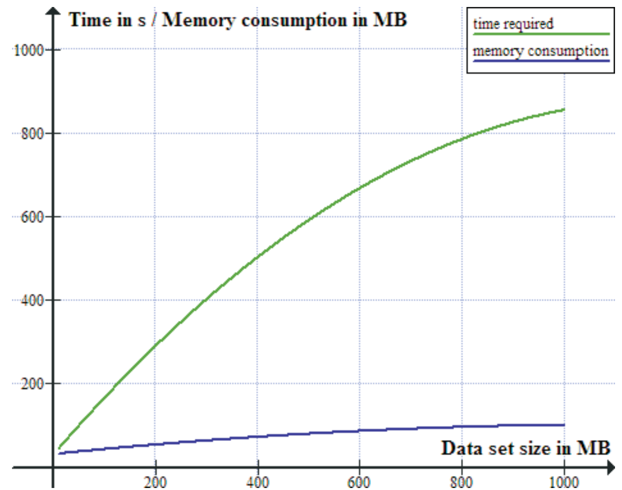

Figure 4. Graph of XStruct scaling to document size regarding time and memory consumption.

all memory occupied by one process and therefore the memory of the virtual machine, too.

However, the key point of the results presented here is the slope of the graph falling with increasing input size for both memory consumption and time requirement. This is clearly an effect of XStruct's behavior to store already generalized element content models efficiently and not to store duplicate element content models. More impressively, we observed that the process of finding and saving the element content models requires even less memory than the factoring algorithm executed afterwards. This is the case due to many recursive calls and additional data structures used during factoring.

5.3 Handling many XML documents as input

We further compared XStruct's behavior when processing many small XML documents vs. processing a single large document. We compare input to XStruct of same size, but different structure. In one case we ran XStruct with a single document of the specified size as input, in the other with 10 documents each one-tenth the size of the single document. In general we expected to see few differences in XStruct's runtime behavior in these two scenarios. However, the results are somewhat surprising, as can be found in Table 2 and Figure 5.

Note that figure 2 compares input to XStruct of same size, but different structure. In the one document case, we ran XStruct with one document of the specified size as input, in the other with 10 documents

[3]www.sysinternals.com

Input size	Time required	Memory allocated
10 x 1 MB	3 s	20 MB
1 x 10 MB	16 s	24 MB
10 x 5 MB	19 s	22 MB
1 x 50 MB	110 s	40 MB
10 x 10 MB	37 s	24 MB
1 x 100 MB	197 s	50 MB
10 x 50 MB	227 s	40 MB
1 x 500 MB	578 s	74 MB

Table 2. Results of the experiments comparing XStruct's runtime behavior with input of a single document and many small documents.

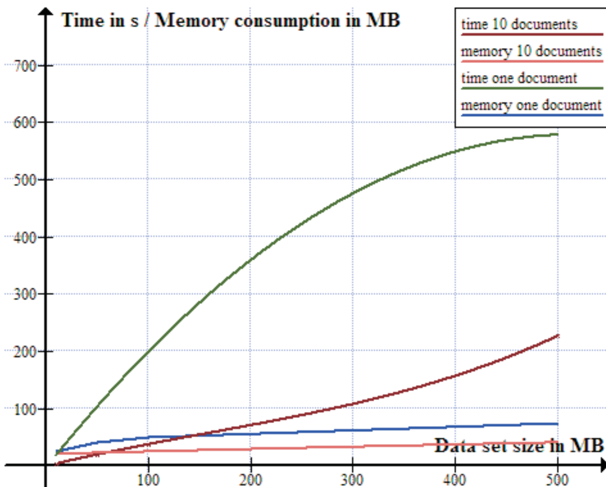

Figure 5. Graph of XStruct scaling to input size of one or many files.

each of one tenth the size of the single one.

For schemata extracted by XStruct it makes no practical difference whether many XML documents are used as input or whether they are all put together into one single document. The different results for different inputs of same size seem surprising at first, but can be explained, when taking into account xmlgen's data generation method. Effectively, the generated ten documents with a small size are identical. XStruct handles this situation as follows: when the first document was scanned, all element content models have been created. When scanning the following files, XStruct firstly constructs the element content models for each element's occurrence, but does not store them afterwards since they have already been stored when scanning the first document. As a result, there are also less element content models to factor for each element than when scanning the single large file, since here we have more element content models for one element, as xmlgen varies the generated data according to a probability distribution. This explains the even better runtime characteristics of XStruct when parsing many small documents compared to parsing one large. This is due to the structure of the generated data and in general XStruct performs quite similarly for input of similar sizes, regardless of whether it is distributed over many documents or condensed in one.

All in all, XStruct is shown to also be able to extract the schema of many XML documents efficiently.

6 Conclusions and Outlook

In this paper we introduced XStruct, an automatic technique for learning the structure of a collection of XML documents by using known algorithms and adapting them to this special problem.

We described what is necessary to extend the ideas of [5] to handle not only single XML documents, but also a collection of XML documents. We furthermore introduced ideas for implementing the algorithms efficiently with respect to time and memory constraints. This comprises data structures that are especially important for ensuring XStruct's scalability, as well as of algorithmic modifications. We further modified the algorithm shown in [4] and so were able to adapt it to suit our needs. Finally, we introduced techniques for determining datatypes of XML elements and attributes and for recognizing attributes. We experimentally showed our ideas to be practical and fulfilling the requirements of scalability in time and memory consumption.

There are several possibilities that could bear potential to improving the algorithms output and are subject to future work:

- Regarding the detection of datatypes, there is much potential for advancing the proposed simple algorithm in many directions. For example one could try to recognize patterns in the data to extract them, too. Or a more advanced version of the method to determine datatypes could be used, that uses sets of possible datatypes for data instead of only giving one type.

- One thing that appears not to be overly convincing is the complexity inherent to the factoring algorithm. Although it is shown in [4] that even this general approach is much faster and simpler than other known algorithms that not only approximate but find the optimal solution for boolean expressions, it remains a complex part of XStruct. We plan to align the yet different data structures of

model extraction module and the factoring to further improve memory consumption.

- If a factoring algorithm would be employed that offers the ability to factor each new model when read during parsing of one occurrence of an element (“on the fly factoring”) this could reduce the memory consumption even further since the element content models would not have to be stored any longer than until factoring is finished.
- As mentioned in this paper, an extension of XStruct to fulfill iterative schema extraction requirements could be readily implemented, in conjunction with the preceding issue even more easily and elegantly.

References

[1] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

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

[3] B. Chidlovskii. Schema extraction from xml: A grammatical inference approach. In *Proceedings of the International Workshop on Knowledge Represenation Meets Databases (KRDB)*, 2001.

[4] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning document type descriptors from xml document collections. *Data Mining and Knowledge Discovery*, 7(1):23–56, 2003.

[5] J.-K. Min, J.-Y. Ahn, and C.-W. Chung. Efficient extraction of schemas for XML documents. *Information Processing Letters*, 85:7–12, 2003.

[6] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 295–306, Seattle, WA, 1998.

[7] C. Romberg. Untersuchungen zur automatischen XML-Schema-Ableitung. Master’s thesis, Universität Rostock, 2001.

[8] Q. Y. Wang, J. X. Yu, and K.-F. Wong. Approximate graph schema extraction for semi-structured data. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 302–316, 2000.