Analysis on Inference Mechanisms for Schema-driven Forms Generation

Josef Spillner, Alexander Schill
{josef.spillner,alexander.schill}@tu-dresden.de

Abstract: Auto-generated graphical user interfaces can save time and avoid mistakes in application development. Data description schemas can form the base for these generation mechanisms. The use of XML-based languages for the schema and for the resulting user interfaces presents a couple of challenges with contemporary representatives of both. We selected XML Schema and XForms as those representatives. Our goal was to explore as many mechanisms as possible which will lead to a complete generated form which can produce or consume data according to the schema. The mechanisms were implemented and are explained in this contribution.

1 Introduction

The model-driven architecture (MDA) advocates a single-source application development methodology where a set of models is sufficient to generate applications or parts thereof. The generation of graphical user interfaces (GUIs) from data schemas is a common pattern within MDA, in particular for database applications, but also increasingly for web service interfaces [Lü05].

Forms in particular are a special type of GUI which can be generated easily. They show several similarities to schemas, such as correspondence of structure and data types.

While numerous description languages for data schemas and user interfaces exist, XML-based languages are popular due to the tool support for transformation and reuse. In particular, XML Schema (XSD) [Fal04] is common in describing data structures and types, and XForms [BLM+06] is often used when describing user interfaces in a declarative manner.

Form inference as the process of transforming XSD to XForms will be the topic of this paper. The structure is as follows: First, general requirements of the transformation process will be presented. Then, the nature of schemas (both in structure and data types) will be given as an introduction, followed by a similar overview on XForms. The main part will then present the challenges of the transformation, primarily concentrating on mapping mechanisms, data models and instances, lists and regular expressions. Finally, statements of applicability on web services and implementation experience is given, before the results are summarised.

2 Transformations

Based on schema information, forms with a certain characteristics can be created. Input forms will produce an instance document which conforms to the schema upon submission. Output forms will take an instance document and display its contents. To many forms in data processing applications these two characteristics apply at the same time.

All forms derived from a schema can have the following quality levels, among others, in increasing order of complexity:

- Type-safety, meaning that input is checked during fill-out of the form and before submission.
- Exploitation, meaning that all possible variants and combinations of instances within the schema constraints can be created.
- Usability, meaning that it is possible to fill out any form without prior knowledge of the form structure.

The interesting questions are centred around the level of quality which can be achieved by solely relying on schema information for form generation. It is already known that it is not possible to describe arbitrary XML instances with XSD. [1] We will show in addition that it is not possible to derive usable or exploiting XForms documents from arbitrary XSD. When taking a few restrictions on the schemas into account, exploiting forms become possible, while usable forms require additional hints and thus remain outside the scope of this paper. Since these restrictions are not imposed by the methods, but rather by the two XML formats under examination, suggestions towards the improvement of the XSD and XForms specifications will appear throughout the paper so that the limitation declared above can be reduced.

3 Introduction to XSD and XForms

Exploring the possibilities of XSD means exploring the structures it can describe and the associated system of extensible types. In particular, the usefulness of each of its quite complex expressions will have to be evaluated. At first, the structure and data types of XSD will be explained, followed by a presentation of relevant parts of XForms.

3.1 Schema structure

XSD is often considered to be complex. A posting to the xml-dev mailing list [Cha02] mirrors a wide-spread understanding of some of its issues:

[1] A trivial counter example will follow.

> The problem is that W3C XSD is "self-balkanizing" – it's clear from this thread that the spec is so complex and obscure that in practice only its most basic features (roughly those it shares with DTDs and RELAX NG plus the basic datatypes) can truly be counted on to be interoperable across implementations and authoring tools, or understandable by any but the most specialized experts.

Research performed at University of Limburg two years later supports this view with a statistical analysis of existing real-world schemas [BNdB04]. This aspect is interesting for the determination of the features of inference mechanisms. To make matters even more complicated, XSD is not of the same expressiveness as DTD or RELAX NG, meaning that one schema format cannot deterministically be transformed into another one of them. Neither of them are sufficient for describing even simple XML documents like the following: An element X has exactly two child elements Y, and carries either attribute A or attribute B.

Pragmatism due to the dominance in web services and other XML processing applications still lets us concentrate on XSD for the remainder of the text, without letting its complexity get into the way. We'll therefore distinguish the contents of the XSD specification which are relevant for inference-based GUI generation from those which are not.

Categorising all XSD elements and attributes results in the following groups: Elements are available for the description of the structure of an XML document, for value constraints through unique keys and simple type definitions, and as short aliases of other XSD elements. Attributes also exist for those three categories but also for object oriented schema usage restrictions and for namespace handling. Some documentation elements could be regarded syntactic sugar but provide valuable information to the inference mechanism if they're actually present. The inclusion of OOP and unique key features, which are mainly of interest for schema authors and instance processors respectively, can be left out for the form inference. All elements and attributes collected in the short aliases and namespaces categories make schemas efficient to read and author, but add nothing to the expressiveness of XSD, and will be left out as well.

For the remainder of the paper, attributes in a schema shall be treated just like elements. There is no need to have them appear differently in forms, as long as the distinction can still be maintained for the form submission.

3.2 Schema data types

XSD has a set of built-in primitive types. These are disjunct types like decimal and float numbers, strings, dates or XML qualified names. Other types may be derived from those. For example, an integer is a specialisation of a decimal number, and an unsigned byte is again a specialisation thereof. Some derived types are part of XSD, others will be part of the schema specification to be used for the description of a particular XML structure, and are created by the respective authors. Derivation happens in form of restricting the possible types by enumerations, minimum and maximum values and patterns, among others.

All of those datatypes are known to be atomic - an instance has exactly one value of its type. By contrast, list datatypes are used for XML elements whose value is a list of atomic types. Furthermore, union datatypes describe a union of one or more datatypes which are either atomic or list datatypes. All of the above, that is, atomic, list and union types, are collectively referred to as simple types. In contrast, complex types contain one or more of such simple types in a sequence of elements and as attributes. XForms selects matching input and output fields automatically for simple types, but doesn't on its own include support for complex types.

3.3 XForms controls

XForms provides four sets of controls: Buttons, input fields, selection fields and output fields. Input controls are `input`, `secret`, `textarea` and `upload`, depending on the semantics of the text to be written into it: Will it be a simple line of text, a password, a text spanning multiple lines or the name of a file which should be attached to the upload data? Selection fields either provide a list to select one or possibly many items from (`select`, `select1`) or a slider representing a bounded set of numerical values (`range`). Finally, the `output` element will simply output text or another representation of data. All output fields are bound to instance data, whereas for all input fields this data initially needs to be created by either the XForms processor or the transformation mechanism. We'll elaborate on this when we get to the mechanisms.

3.4 Type system incompatibilities

The schema language understood by XForms is XSD. However, some subtle differences exist which might make it hard or impossible to use arbitrary XSD-specified schemas in XForms document.

The XSD datatype `duration` has been replaced by the two XForms types `dayTimeDuration` and `yearMonthDuration`. This needs an unfortunate extra check and possible conversion in the inference mechanism.

Simple list datatypes can be bound to form controls to let them produce lists of a particular base type. However, current implementations do not provide separate UI controls for list types, leaving the decision to the inference mechanism to either implement variable-size lists (as is being done for elements of varying count in complex types already), or to have a generic input field where the user fills in the values, hopefully knowing to separate them by a space character.

The XForms specification has faced this problem by the introduction of the `listItem` type which is a base type for lists not allowing space characters as part of its value. It is probably a good idea for the inference mechanism to rewrite all `xsd:string`-based lists to use this type.

4 Mapping mechanism

A generic mapping is such that any simple type may render to an unrestricted input field. This will however not gain any type safety, and certainly will suffer from typographical errors by the user. It will also not honour the number of XForms controls, which is certainly not vast but it's flexible enough to create better GUIs than those with just plain input controls.

It should be mentioned that formal schemas are not generative on their own. A good example in XSD is the `any` type, which doesn't describe its content other than that it has to be XML-formatted. Despite such drawbacks, XSD facilities provide many clues to how the GUI should appear.

The XSD restrictions `minInclusive`, `maxInclusive`, `minExclusive` and `maxExclusive` are of interest for ordered values. If both a minimum and a maximum facet is given, an XForms `range` widget may be put into place. This is easy for the `*Inclusive` facets, as they will map directly to the `start` and `end` attribute of the control. However, the `*Exclusive` facets are hard to deal with. For example, a date is an ordered type. This means that the inference mechanism has to know how dates are built, while the XForms client implementation needs to know it as well. The usage of the `*Exclusive` facets must therefore be considered a limiting factor for GUI generation.

Enumeration restrictions simply map to `select1` controls. Specifically, if it is an open enumeration (a union of a restricted type and the unrestricted type itself), the control will provide an `open` selection, allowing the user to input any value but still providing the enumerations as convenient default values.

Lists and unions can be bound to XForms controls, however there is no indication to the user about the underlying data structure. A way to solve the problem would be for lists to be displayed as a list of several controls, which can be resized as needed. However, due to the one-to-one mapping of XForms controls to XSD elements, this is not entirely possible.

A limited workaround is to create one input control for each element, and an additional read-only input control which calculates its content automatically based on what is filled into the other controls. The XPath `concat()` function can be used for this task. It does however only work with a known fixed number of controls, and therefore with lists of fixed size, or with some more work at least a known maximum size. The XPath `calculate()` expression for the read-only control will be of the form `concat(../input[1], ' ', ../input[2] ...)`. The composition can be seen in figure 1. It must be ensured that the data inside the helper controls will not become part of the generated instance data. A way to do this is to set the `relevant` property to the XPath expression `false()`.

Unions can be edited by adding a type-toggle button: A button which switches the input control among those which are mandated by the member types of the union, e.g. a range control and an input control (figure 2). The XForms `choice` expression is suitable to achieve type-toggling.

For numerical input, it would be handy if there was a numeric input widget which allowed integer or floating point numbers only. Absence of such a control requires decent usage

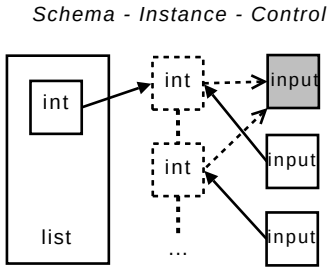

Figure 1: Input method for simple lists

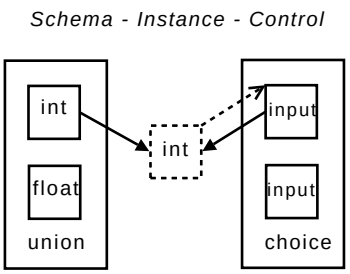

Figure 2: Input method for unions

of CSS rules to indicate invalid input, e.g. by colouring the input field accordingly. According to [Dub05], creating forms should always happen with the inclusion of *user-space* representation of values in mind, in addition to the lexical space and the value space of types. Following this suggestion, the mapping mechanism can help finding more suitable form control representations.

4.1 XForms models

While the statements above referenced the XForms form controls, each XForms instance, which may not be free-standing but must be embedded into a host document instead, has to specify the model it works on. In XHTML, the form controls can be spread freely in the `<body>` tag, whereas the model is usually kept in the `<head>` tag.

A model contains an initial instance of the schema, the schema by itself (should this be necessary due to custom datatypes), optional binding expressions to constrain the way a form is filled out and a target to submit the form to, called the submission element. The initial instance is explained in the XForms specification, and there is a notion of an empty instance which is then filled out by the user through the form controls. This will not work in practice, however, due to restricted datatypes. For any datatype which does not permit the empty value to be a legal value, omitting the initial instance or using an empty instance will produce XForms validation errors.

In the inference mechanism, *smart* initial instance data is used to solve this problem at least syntactically. Examples of creation rules are:

- Boolean types may only be true or false.
- Numbers cannot be represented by empty strings.
- Enumerations may only have one of their values.
- Elements must occur at least as many times as mandated by their `minOccurs` attribute.

While the XSD specification already defined canonical lexical representation, what is needed here is a canonical lexical representation of a neutral value in relation to a common operation such as addition or concatenation, or, in the absence of such one, one value out of the value space by convention. Therefore, the *smart* value for integers will be 0, for booleans $true$ and for enumerations it will always be the first of their values.

4.2 Value constraints

Simple types constrained with regular expressions need a smart value which matches said expression. The grammar of such expressions in XSD is based on the well-known one from the programming language Perl, however the differences are plenty: Character range intersections were introduced (`[\d-[:]]`), the match region parameters (anchors) `^` and `$` were removed, and new character classes were added to match XML production rules, like `\c\i`. Within a simple type definition, minimum and maximum string length constraints might exist which further limit the set of productions yielded by the regular expression. What is needed is to calculate one value for each term quantifier so that the sum of the length of all terms fits the constraints. We have developed the RegExpInstantiator class to take care of this task.

To summarise, here are the rules to produce valid instances based on the type and the restrictions:

- Enumerations, no matter what their type is, already provide at least one instance described by an `enumeration` facet. It is still possible to detect type mismatches when checking all enumerations against the restrictions of the types further down the type hierarchy, but this is a schema checker task and doesn't have to be done at run-time for schemas considered valid.
- Strings are described by a `pattern` facet, and constrained by `minLength`, `maxLength` and `length` facets. It is deterministically possible to create an initial instance or to detect the impossibility to do so.
- Numerical values are constrained in range by several facets, namely `minInclusive`, `maxInclusive`, `minExclusive` and `maxExclusive`. It is deterministically possible to find one value within the allowed range, or to detect the impossibility to do so.
- The white space facet `whiteSpace` can be ignored for instance generation.

4.3 Lifting of regular expressions

While the previous sections treated regular expression facets like value-restricted strings, it is also possible to make editing them more comfortable.

Similar to complex data types, regular expressions consist of a sequence of terms, each one having a certain minimum and maximum occurrence. Likewise, nested expressions are possible similar to how elements within a complex type can be of a complex type again.

Therefore, it is suggested by us to convert any regular expression into a schema, and thus make it possible to use the presented inference rules to generate a GUI from it. The advantages include the ability to use concepts such as help texts for the individual terms, the distinction between editable terms and fixed ones, and type safety for each typed-in character, not needing to evaluate the resulting value against the whole expression on every such event.

As an example, the expression `\d\d?\.\d\d?\.(18|19|20)\d{2}` describes valid birth dates, like *5.10.1881* of Pablo Picasso or *22.6.1919* of Konrad Zuse. Syntactically, leading zeros are not required for the day or month, but still for the year of a century, which is what most people would intuitively use. The author of the expression intended to limit the centuries to maybe express that only records from these years are stored in the database, which is fine.

The conversion algorithm checks for branches as in the `(18|19|20)` term. If the branches are fixed, an enumeration is created, or otherwise a choice in the schema. If nested terms are encountered (not in the example), new complex types are generated from them. The example expression would be expressed in XML Schema as follows:

```
<xsd:element name="expression">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="day" type="digittype"/>
      <xsd:element name="day2" type="digittype"
                   minOccurs="0"/>
      <xsd:element name="dot" fixed="."/>
      <xsd:element name="month" type="digittype"/>
      <xsd:element name="month2" type="digittype"
                   minOccurs="0"/>
      <xsd:element name="dot2" fixed="."/>
      <xsd:element name="century" type="centurytype"/>
      <xsd:element name="year" type="digittype"
                   minOccurs="2" maxOccurs="2"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:simpleType name="digittype">
  <xsd:restriction base="xsd:int">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="9"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="centurytype">
```

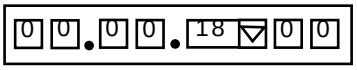

Figure 3: Form generated from regular expression

```
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="18"/>
      <xsd:enumeration value="19"/>
      <xsd:enumeration value="20"/>
    </xsd:restriction>
  </xsd:simpleType>
```

Note that the element names were chosen in a way that makes understanding easier. For automatically generated schemas, they would probably just follow some random naming convention.

The resulting form would schematically look like the one shown in figure 3. As can be seen from the filled-in numbers, the regular expression is matched, although the date makes no sense semantically. Therefore, the regular expression could be improved [2], or other means such as semantic hints are needed.

Of course, form fragments generated from this kind of schema would have to collapse their instance data to one single value, similar to how finite simple lists were handled.

4.4 Complex lists

One structural feature of XSD is to be able to specify lists of elements with a certain minimum and maximum number of occurrences. In the trivial case, both are set to 1, which is the default. When the minimum occurrence is 0, the element is made optional. When the maximum occurrence is greater than the minimum occurrence, a list of elements emerges. In XForms, lists are expressed by using `xf:repeat` structures. The mapping of XSD lists to XForms lists presents a couple of issues when the user is allowed to change the number of elements which is in accordance with the schema.

The first issue becomes apparent when the list is allowed to have zero elements and the user removes all of them. In that case, no XForms instance data is left. However, instance data is needed to copy it for further inserts. In XForms 1.1, this problem has been solved.

The second issue is related to the first: A user would expect new entries to be empty at the beginning. However, in XForms 1.0, the copy of the instance data would always copy the last item of the instance data. In XForms 1.1, the origin of the new instance item can be configured by using the `origin` attribute.

[2] For the month, the expression `(0?[1-9]|1[012])` would result, whereas for days, the maximum number depends on the month and year. Not recommended.

The third issue is to restrict the number of items to be no smaller than and no higher than what the schema allows. Again, XForms 1.1 provides a solution for this problem, namely the `#action-conditional` construct, whereas in XForms 1.0 such a restriction is not directly possible. However, a workaround exists for XForms 1.0. The solution is to introduce a second schema and instance within the default model. The schema is copied from the existing one. Every repeated item is then first converted to a complex type, if not already the case, and then receives two dummy child elements for addition and deletion. Binding these to the number of occurrences of the repeated item in the original instance makes it possible to restrict the number of list items according to the schema. Upon submission, the second instance will simply be ignored. This workaround would lead to the following XForms structure for a repeatable item located at `/parent/item`:

```
<xf:model>
<xf:bind nodeset="instance('lists')/item/add"
         relevant="count(instance('main')/item) &lt; 5"/>
</xf:model>
...
<xf:group ref="instance('lists')/item/add">
<xf:insert nodeset="instance('main')/item" at="last()"
           position="after()"/>
</xf:group>
```

5 Application: GUI generation for web services

XML Schema is often used in web service description files to give normative information about the nature of input and output messages of its services. The mechanisms found in this paper apply to web services as well. For each service operation, input and output forms can be generated. An approach for interactive ad-hoc service invocation tools which includes the inference mechanisms is known to be *Web Services Graphical User Interfaces* (WSGUI) [SBS07].

Our WSGUI implementation called Dynvoker covers all of the presented mechanisms. Along with example files highlighting the issues presented in the previous chapters, it is offered for download and examination [SB+07].

The work on Dynvoker has motivated once again to have a formal look on a mapping between XML Schema and XForms to move towards dynamic use of services by humans.

6 Related work

Schema-driven forms generation has been the topic for publications for a long time. Even the specific transformation of XSD to XForms has also been analysed already. However, most of the approaches use a pure XML transformation by way of XSL stylesheets

[Moe06], while our approach doesn't mandate such an algorithm.

A sophisticated composite schema transformer [GF03] highlights the issues with XSLT. While we were able to confirm their basic methodology of processing the schema twice, for the XForms model and the form controls, we were not satisfied with the idea of working directly on the XSD syntax. Instead, our XSD parser presents a logical view on schemas. This approach transparently supports composite schemas as well. A recently published thesis [Mic07] provides a similar solution by extending XSL. Future XSL-based transformers might incorporate this work and thus overcome their current limitations.

Schemas other than XSD have been proposed for XForms generation in [LK05] and implemented in *XForms-Generator*. Due to the nature of DTD, a number of problems arise. First, no namespace-aware XML handling is possible. Since XForms itself uses (modified) XSD to describe its data, DTDs must be converted to XSD as an extra transformation. An additional disadvantage is that no natural integration with web services arises. Finally, the paper didn't consider restricted lists.

Using XSD, but producing different output formats, has also been considered [LLK04]. The analysis of XSD transformation options such as types, sequences, repetition and alternatives is rather complete in this work. Unfortunately, no public implementation is available.

A few additional proprietary implementations exist [KKWS06]. Since neither source code nor publications exist on those, it is not clear if the mechanisms used by them are equivalent to those presented in this paper.

7 Conclusions

Schema-driven XForms generation has been explored in detail in this paper. We have shown that several obstacles could be overcome, while others (like dynamic simple lists) could not. The transformation mechanisms for restricted lists, unions and regular expressions can nevertheless be considered sufficient for a lot of XSD schemas found in practice. The usefulness of the mechanisms could be increased by appropriate changes to the XSD and XForms specification.

An implementation named Dynvoker has been made available which uses inference to call web services dynamically. It should be pointed out for completeness that inference also applies to extracting information from WSDL files for the creation of navigational elements in addition to inference from XSD. However, since no interaction elements are affected, this topic has been omitted from this paper.

As a closing remark, it should be mentioned that effective research on this topic requires working XForms processors. Even several years after the first standardisation, many issues were found with existing implementations during our work. We conclude that the adoption of XForms could be vastly extended by improved implementations. Our research has already uncovered some bugs in one of them, and will likely break out more when we look at advanced topics such as device adaptation.

Acknowledgements

I would like to thank Steve Speicher and Aaron Reed for their input on XForms-related questions, and Alexander Lorz and Iris Braun for some helpful comments.

References

[BLM+06] John M. Boyer, David Landwehr, Roland Merrick, T.V. Raman, et al. *XForms 1.0 (W3C Recommendation)*, 2nd edition, March 2006. `http://www.w3.org/TR/xforms/`.

[BNdB04] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML Schema: A Practical Study. In *Proceedings*, June 2004. Seventh International Workshop on Web and Databases (WebDB 2004), Paris, France.

[Cha02] Michael Champion. XML-Dev mailing list, June 2002. `http://www.xml.org/xml/xmldev.shtml`.

[Dub05] Micah Dubinko. *XForms Essentials*, chapter 4. O'Reilly, 2005.

[Fal04] David C. Fallside. *XML Schema Part 2: Datatypes (Second Edition)*, 1.0 edition, October 2004.

[GF03] Patrick Garvey and Bill French. Generating User Interfaces from Composite Schemas. December 2003. XML 2003, Philadelphia, USA.

[KKWS06] Kevin E. Kelly, Jan Joseph Kratky, Keith Wells, and Steve Speicher. XML Forms Generator, March 2006. IBM alphaWorks.

[LK05] Eunujung Lee and Tae-Hoon Kim. Automatic Generation of XForms Code using DTD. Jan 2005. ACIS International Conference on Computer and Information Science (ICIS).

[LLK04] Patrick Lay and Stefan Lüttringhaus-Kappel. Transforming XML Schemas into Swing GUIs. September 2004. Web Applications and Middleware workshop (WAM2004), Ulm, Germany.

[Lü05] Tim Lüecke. Development of a Concept for Creating and Managing User-Interfaces bound to Business Processes. Master's thesis, Universität Hannover, 2005.

[Mic07] Felix Michel. Representation of XML Schema Components. Master's thesis, ETH Zürich, Dept. of Electrical Engineering and Information Technology/University of California, Berkeley, 2007.

[Moe06] Joerg Moebius. xsdTransformer, November 2006. `http://xsdtrans.sourceforge.net/#xsd2fx`.

[SB+07] Josef Spillner, Nicolás Bleyh, et al. Dynvoker web service invocation servlet, Feburary 2007. `http://dynvocation.selfip.net/dynvoker/`.

[SBS07] Josef Spillner, Iris Braun, and Alexander Schill. Flexible Human Service Interfaces. volume HCI, pages 79–85, June 2007. International Conference on Enterprise Information Systems (ICEIS), Funchal, Madeira - Portugal.