

STAX/bc: A binding compiler for event-based XML data binding APIs

Florian Reuter, Norbert Luttenberger

Communication Systems Research Group
Christian-Albrechts-University in Kiel, Germany

{flr|nl}@informatik.uni-kiel.de

ABSTRACT

In this article we present the STAX/bc¹ binding compiler which generates event-based data binding APIs out of W3C Schema descriptions. The event-based nature of the generated data binding APIs allows programmers the development of:

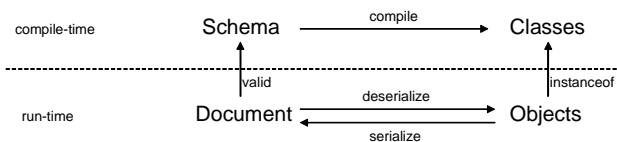
- applications for processing large XML documents or XML data streams,
- applications for resource constraint systems without enough space to deserialize XML documents into main memory, and
- applications which must have full control about the data structures used.

STAX/bc binding compiler can be written targeting any object oriented language, like C++, Java or C#, e.g.

1. Introduction

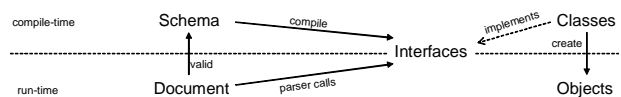
For the development of XML-processing applications powerful XML-APIs are needed which allow programmers to concentrate on the application logic rather than on the mapping between XML documents and programming language objects.

Data binding frameworks provide a good way to achieve this *deserialization-* and *serialization* mapping of XML documents to programming language objects resp. vice versa instead of using low-level APIs like DOM and SAX. Data binding frameworks provide a *binding compiler* which, given a grammar of the XML documents as input, produces a set of classes together with deserialization and serialization functions, which transform XML documents into instances of the generated classes and vice versa. Current data binding frameworks use schema languages like DTDs, W3C Schema and RelaxNG to describe the document grammar. The following figure shows the functioning of data binding frameworks graphically:



¹ STAX is a four letter acronym for Simple *Typed* API for XML, which alludes to the event-based nature of its namesakes SAX (Simple API for XML) and StAX (Streaming API for XML) but stresses the fact that STAX is *Schema-typed*. To avoid confusion with StAX we will write STAX/bc for the presented binding compiler.

The STAX/bc binding compiler presented in this paper works differently. STAX/bc shares the idea with data binding frameworks of using a binding compiler which takes a schema description as input, but the STAX/bc-binding compiler does not produce classes. Instead, the STAX/bc-binding compiler produces interfaces between the application logic and the XML parser. By implementing these interfaces an application can create *any kind* of objects at runtime out of the parsed XML document. The following figure shows the functioning of STAX/bc schematically:



This concept allows the development of

- applications for typed processing of XML documents or XML data streams,
- applications for resource constraint systems without enough space to deserialize XML documents into main memory,
- applications which must have full control about the data structures used.

2. Related Work

Currently available data binding frameworks like the W3C schema based XSD[28], JAXB[20,9], Castor[21], jBIND[22], gSOAP[2], the RelaxNG-based Relaxer[10] or the DTD-based Zeus[23,9] or Quick[24,9] generate classes together with deserialization and serialization functions as shown above. These classes explicitly define the in-memory representation of the deserialized XML documents.

When an XML document is parsed, the whole XML document is deserialized into main memory. This is problematic for large XML documents and impossible for XML streams.

Another disadvantage of the previously mentioned data binding frameworks is that the programmer has no, or little, control about the data structures in which the information from the XML documents is deserialized. Some data binding compiler apply design pattern like factory e.g. or customization options to overcome this, but for example the deserialization of XML documents directly into Java/Swing components as shown in chapter 8 of this paper is not possible with current data binding frameworks.

Event-based XML-APIs generate a sequence of events while parsing XML documents. These events then can be handled by an

application which allows an application to process XML documents on-the-fly. A well known event-based API is SAX. The SAX-API reports XML Infoset-like events to the application via the `ContentHandler` interface. Applications can handle the events by implementing this interface.

The difference to its namesake STAX/bc is that SAX reports XML Infoset-like events via a single interface `ContentHandler`, whereas the STAX/bc binding compiler generates interfaces out of a schema description. Informally spoken, SAX generates “untyped” events whereas STAX/bc generates “typed” events.

The advantages of event-based APIs are, that XML parsers with event-based XML-APIs can be implemented very resource sparing, since they only generate events and do not have to load the whole XML document into main memory. Furthermore programmers can, by handling the events in an appropriate way, store the information from the XML documents in data structures of their choice or process the information on-the-fly.

Beside the mentioned data binding frameworks there exist several other kinds of data binding XML-APIs. Another class of XML-APIs are used in data binding frameworks like [1,18,21]. Here a binding function explicitly defines how XML documents should be bound to programming language objects. E.g. a XML-document `<member><name>Mr. X</name></member>` can be bound to the class `Member{String name;}`; by the following binding function: `<member>→Member, <name>→Member.name` without using an intermediate schema. Often this binding function is (semi-)automatically inferred by the binding framework. These kinds of XML-APIs are favorably used, when “real” semistructured data – e.g. data for which no schema is available or can’t be specified for some reasons – must be processed[1].

Yet another class of XML-APIs are used in XML programming languages like [5,14, 30]. XML programming languages are based on XML Infoset-like structures and they use schemas to perform XML type checking[13].

To the best knowledge of the authors STAX/bc is the first data binding compiler which generates event-based XML data binding APIs out of W3C schema descriptions. Herewith STAX/bc combines the advantages of data binding frameworks and event-based XML-APIs, which are:

- processing of large XML documents and data streams as well as on-the-fly processing,
- free choice of data structures used to maintain the data, and
- easy access to the information within your programming language due to specially tailored interfaces.

3. STAX/bc’s concept

STAX/bc’s technical concepts can be summarized as follows:

- Every W3C schema type is mapped into an interface. By implementing these interfaces an application can bind itself to a STAX/bc-aware XML parser.
- The interfaces own `create` and `add` methods which are called at runtime by the STAX/bc-aware XML parser.

- The `create` method plays the role of a factory method with which the XML parser creates new objects. The `add` method is used by the XML parser to report the generated objects to the application logic.

Since STAX/bc deals only with interfaces and makes no assumptions about the programming languages primitive types, the STAX/bc binding compiler can be used to generate code for any object oriented programming language.

We will first present in chapter 4 the STAX/bc-binding compiler which maps W3C schema descriptions to the interfaces. Thereafter we explain in chapter 5 how the generated interfaces are invoked by the STAX/bc-runtime system. In chapter 6 we present a slight modification of the basic rules given in chapter 4 and 5 which lead to a structural improvement of the generated interface structure. In chapter 7 we address the naming problem and chapter 8 shows the STAX/bc programming exemplarily.

4. STAX/bc-Binding Compiler

This section describes the STAX/bc-binding compiler. As mentioned above the binding compiler is responsible for mapping W3C schema descriptions into interfaces.

The STAX/bc-binding compiler applies four mapping-rules, which are sufficient to map any entirely named W3C schema description into interfaces. A W3C schema description is entirely named if it does not contain neither an anonymous complex type definition, nor an anonymous simple type definition, nor an anonymous model group. This can be achieved for any W3C schema by applying a naming preprocessor. This step is explained in section 7. In the following we assume that the input W3C schema is entirely named.

The mapping-rules are shown in figure 1. We will discuss them briefly now.

Rule 1: Built-in types:

The W3C Schema built-in types, as defined in W3C specification, are mapped to the interfaces bearing the same name as the built-in type suffixed by `Creator` as shown in figure 1.

Any simple type inherits the `addContent(content:char[])` method from the `AnySimpleTypeCreator-Interface`.

The interface representing the W3C Schema built-in type `QName` additionally has an `addNamespaceBinding(prefix:char[], namespace:char[])` method, since `QNames` need access to the actual prefix-namespace mapping.

Rule 2: Top level attribute and element declarations:

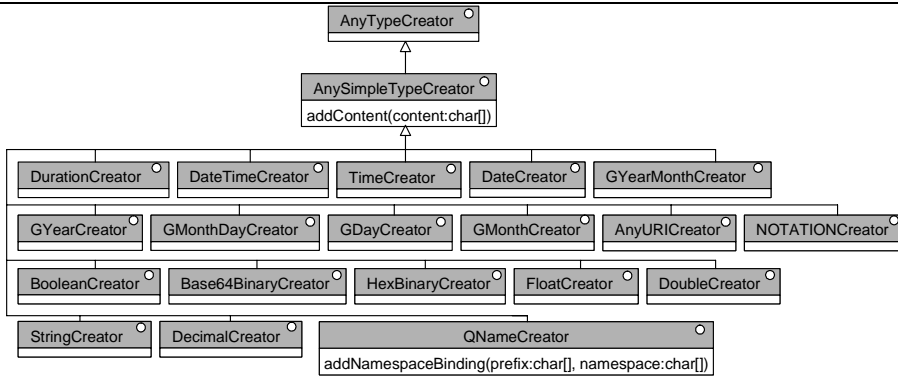
Each schema definition is mapped into an interface named `DocumentCreator`. For every top level particle a `create` and an `add` method are added to the `DocumentCreator` interface.

The `DocumentCreator` interface represents a whole XML document and is always generated. It serves as an entry point for the XML parser.

Rule 3: Type definitions, named model groups, attribute groups:

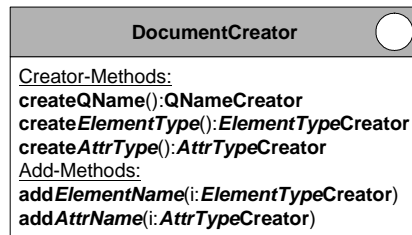
A named complex type with complex content, a named simple type, a named model group and an attribute group definition are mapped into an interface with the type name suffixed by `Creator`:

Figure 1: The STAX-binding compiler's mapping rules:



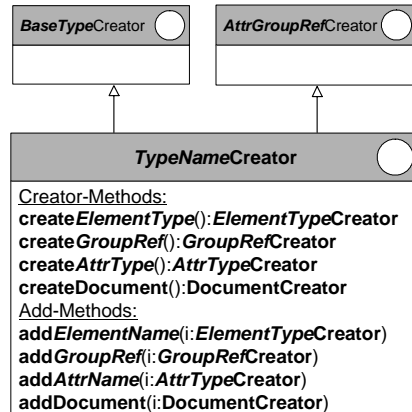
Rule 1. Built-in types.

```
<schema>
  (<element name="ElementName" type="ElementType"/>
  |<attribute name="AttrName" type="AttrType"/>
  |...)*
</schema>
```



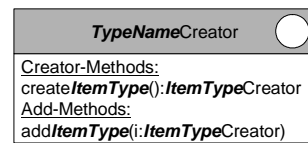
Rule 2. Top level element and top level attribute declarations.

```
<(complexType|simpleType|group|attributeGroup) name="TypeName">
  (<(complexContent|simpleContent)>
  <(restriction|extension) base="BaseType">?
    ((<(all|sequence|choice)>)?
      (<element name="ElementName" type="ElementType"/>
      |<group ref="GroupRef"/>
      |<any ...>)*
    </(all|sequence|choice)>)?
    (<attribute name="AttrName" type="AttrType"/>
    |<attributeGroup ref="AttrGroupRef"/>
    |<anyAttribute .../>)*
  </(restriction|extension)>
  <(complexContent|simpleContent)>)?
</(complexType|simpleType|group|attributeGroup)>
```



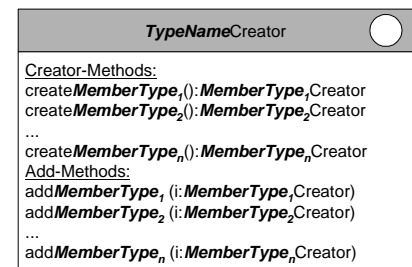
Rule 3. Type definitions, named model groups and attribute groups

```
<simpleType name="TypeName">
  <list itemType="ItemType"/>
</simpleType>
```



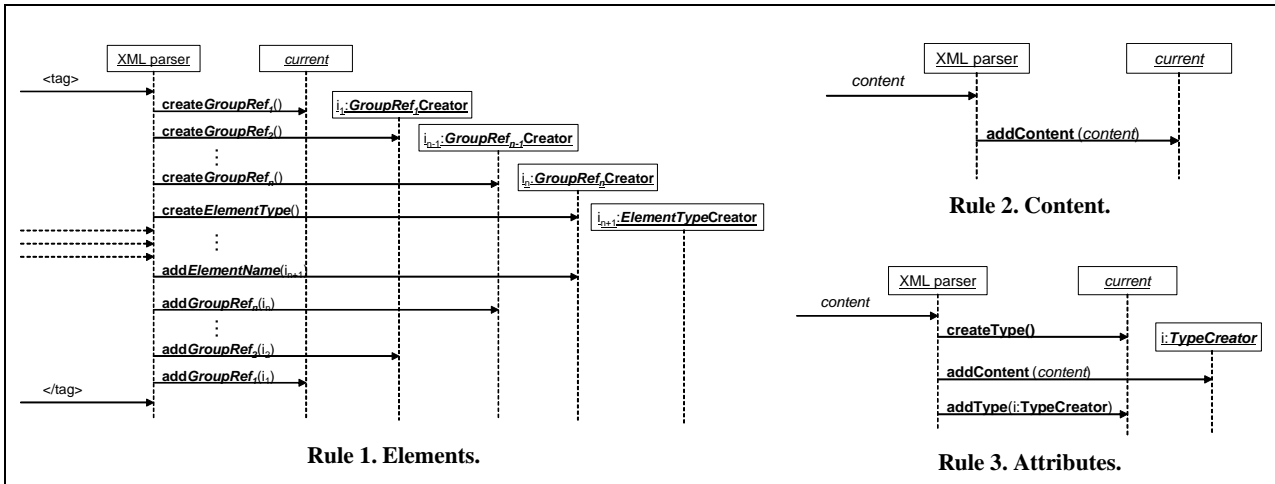
Rule 4. Simple type lists.

```
<simpleType name="TypeName">
  <union memberTypes="MemberType_1 ... MemberType_n"/>
</simpleType>
```



Rule 5. Simple type unions.

Figure 2: The STAX-runtime system:



W3C schema construct	mapped to
<complexType name="TypeName" >	Interface <i>TypeNameCreator</i>
<simpleType name="TypeName" >	
<group ref="TypeName" >	
<attributeGroup name="TypeName" />	

A derivation is mapped to a corresponding generalization:

W3C schema construct	mapped to
<extension base="BaseType" >	Generalization. from <i>BaseTypeCreator</i> to <i>TypeNameCreator</i>
<restriction base="BaseType" >	

Particles and attribute declarations are mapped into create and add methods:

W3C schema construct	mapped to
<element name="ElementName" type="ElementType" >	create <i>ElementType</i> resp. create <i>GroupRef</i> resp. create <i>AttrType</i>
<group ref="GroupRef" >	and
<attribute name="AttrName" type="AttrType" >	add <i>ElementName</i> resp. add <i>GroupRef</i> resp. add <i>AttrType</i>

An attribute group reference is mapped into a generalization:

W3C schema construct	mapped to
<attributeGroup ref="AttrGroupRef" />	Generalization from <i>AttrGroupRefCreator</i> to <i>TypeNameCreator</i> .

The wildcards <any> and <anyAttribute> are mapped into a createDocument and an addDocument method.

Rule 4: Simple type lists

A simple type which defines a list is mapped into an interface with appropriate add and create methods:

W3C schema construct	mapped to
<simpleType name="TypeName" >	Interface <i>TypeNameCreator</i> with create <i>ItemType</i> and add <i>ItemType</i> .
<list itemType="ItemType" >	

Rule 5: Simple type unions

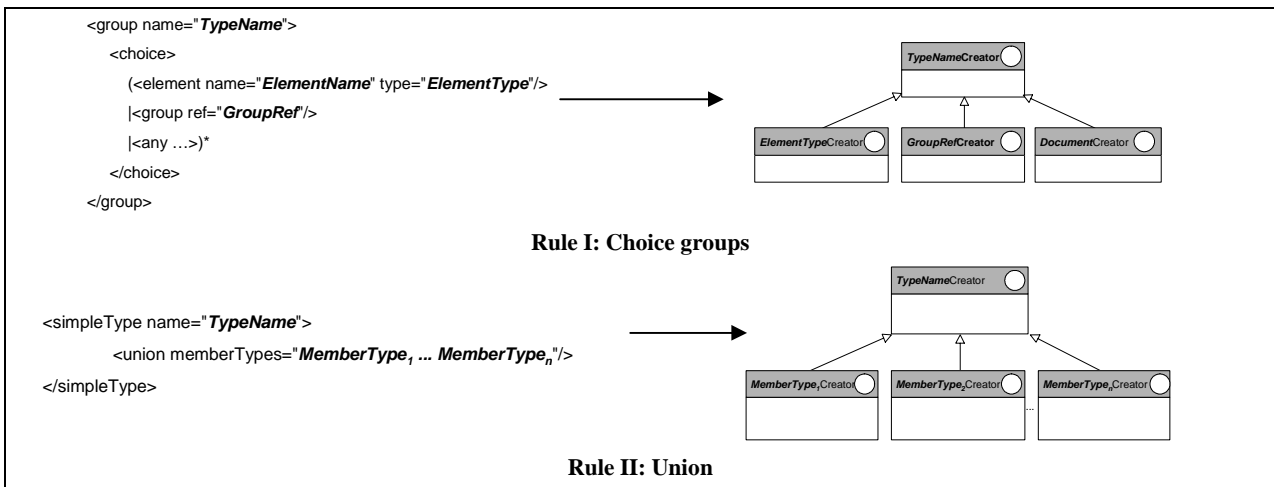
A simple type which defines a union is mapped into an interface with appropriate create and add methods:

W3C schema construct	mapped to
<simpleType name="TypeName" >	Interface named <i>TypeNameCreator</i> with create <i>MemberType_i</i> and add <i>MemberType_i</i> methods.
<union memberType="MemberType ₁ ... MemberType _n " />	

The above rules cover all W3C schema key concepts namely <schema>, <element>, <group>, <attribute>, <simpleType>, <complexType>, <simpleContent>, <complexContent>, <restriction>, <extension>, <all>, <sequence>, <choice>, <any>, <anyAttribute>, <list>, <union>, <attributeGroup>, <enumeration>, <pattern>, <any>, <anyAttribute>. However, there is one unsupported special case: Element declarations of type anyType (e.g. <element name="name" type="xsd:anyType"/>). This is, because anyType elements allow "real" semistructured data, which means that a priori no classes can be generated.

The create-Methods install an abstract factory into every interface which arose from a complex type definition. This gives every implementing class the chance to define the type of its children.

Figure 3: Statically structure improving rules:



An alternative would be the usage of a single global factory. This would remove the create-Methods from the generated interfaces and thus would reduce the number of methods an implementing class must implement. We choose not to use a single global factory, since we want to give parent classes full control over their child types.

5. STAX/bc-Runtime System

In this section we describe the STAX/bc-Runtime system which builds upon a validating W3C Schema-aware XML parser. Depending on the currently parsed XML information item, the runtime system invokes appropriate create and add methods.

A W3C schema is a single-type tree grammar which can be parsed by an event-based deterministic top-down tree automaton[15]. Consider the following algorithm:

1. Upon every start element `<tag>` we differentiate two cases:
 - a. We are currently at the root state. We then search for a top level element declaration `<element name="tag" type="T"/>` and push `T` onto the stack.
 - b. The stack is non-empty. Let `T` be the stack's topmost type. We search for a `<element name="tag" type="T_tag"/>` declaration (resp. `<element ref="tag">` together with the top-level element declaration `<element name="tag" type="T_tag"/>`) reachable via none or more `<group>`s in `T`'s complex type definition and push `T_tag` onto the stack.
2. Upon every end element tag `</tag>` we check the sequence of child types `T1, ... Tn` against the regular expression defined by the current type.
3. Simple content `c` is checked according to the simple type definition.
4. Attributes `name="value"` are checked by searching an appropriate `<attribute name="name" type="T"/>` declaration in the current type and then checking the value using the type `T`.

The above algorithm can easily be extended to produce the STAX/bc create and add events by applying the following rules:

Rule 1: Elements

For every starting tag `<tag>` validated by the schema constructs `<group ref="GroupRef1" />...<group ref="GroupRefn" /><element name="ElementName" type="ElementType"/>` in step 1 of the above algorithm the runtime environment generates the following create methods: First every group involved in the validation of `<tag>` is created using the appropriate create function. Then the complex type associated with the tag is created by the appropriate create function.

After the tag's attributes and children have been processed in the same manner, the created instances are added in step 2 of the above algorithm in reverse order to the belonging parent instance by performing the appropriate add calls. Figure 2 illustrates the invocation mechanism.

Rule 2: Content

For every content event `content` processed in step 3 of the above algorithm the runtime system uses the `addContent` method to pass the incoming content to the simple type (resp. complex type with simple content) instance.

Rule 3: Attributes

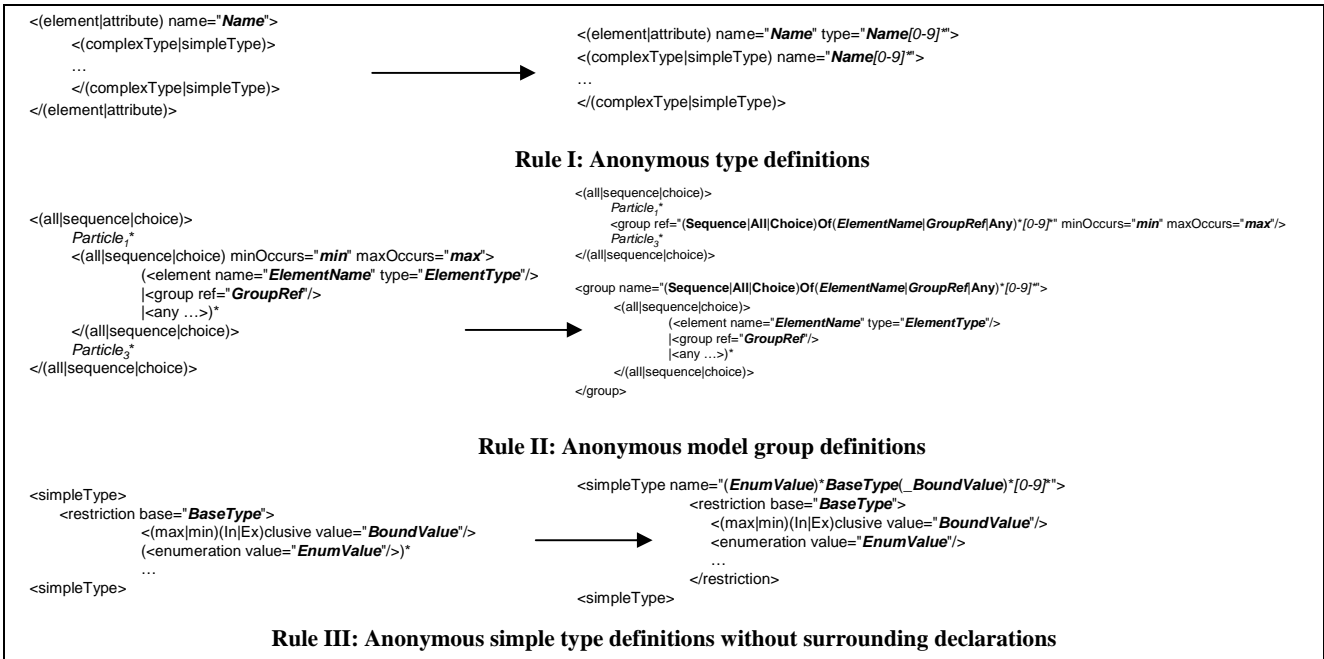
For every attribute `name="value"` processed in step 4 of the above algorithm the runtime system first creates the appropriate simple type by calling the appropriate create method, and then uses the `addContent` method to pass the incoming content to the simple type instance. Finally the simple type instance is added to the parent instance by using the appropriate add method.

6. Improvement of the static structure

There exist two W3C schema constructs for which only one of more given possibilities can occur: The `<choice>` and the `<union>` constructs.

With the previously presented mapping rules these constructs would be mapped to interfaces in a way that this "either ... or" property gets lost.

Figure 4: Naming Rules:

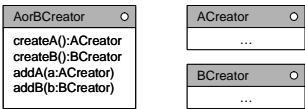


For example consider the schema fragment

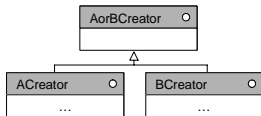
```

<group name="AorB">
  <choice>
    <element name="a" type="A"/>
    <element name="b" type="B"/>
  </choice>
</group>
  
```

It would be mapped to the following interfaces by a non-improved binding compiler:



A much better design is the usage of the OOP concepts polymorphism and inheritance to express the "either ... or" relationship:



This idea is close to the façade design pattern, which provides a unified interface for a set of interfaces in the subsystem[11].

We will now show how the rules from section 4 must be altered in order improve the static design of the generated interfaces.

We highly recommend the usage of these improvements. In all our tests the improvements lead to an interface structure, which was more understandable and easier to program with.

The rules to improve the static structure are shown in fig. 3. We will discuss them briefly now:

Rule I: Choice groups

Model Group definitions whose variety is a disjunction are mapped into an empty interface and a generalization is added to all particles' types.

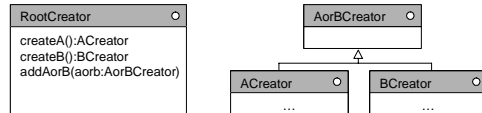
Furthermore every createTypeName method in referencing interfaces is replaced by the create methods which would have been generated by the default rule.

Consider for the example the following schema fragment:

```

<complexType name="Root">
  <group ref="AorB"/>
</complexType>
<group name="AorB">
  <choice>
    <element name="a" type="A"/>
    <element name="b" type="B"/>
  </choice>
</group>
  
```

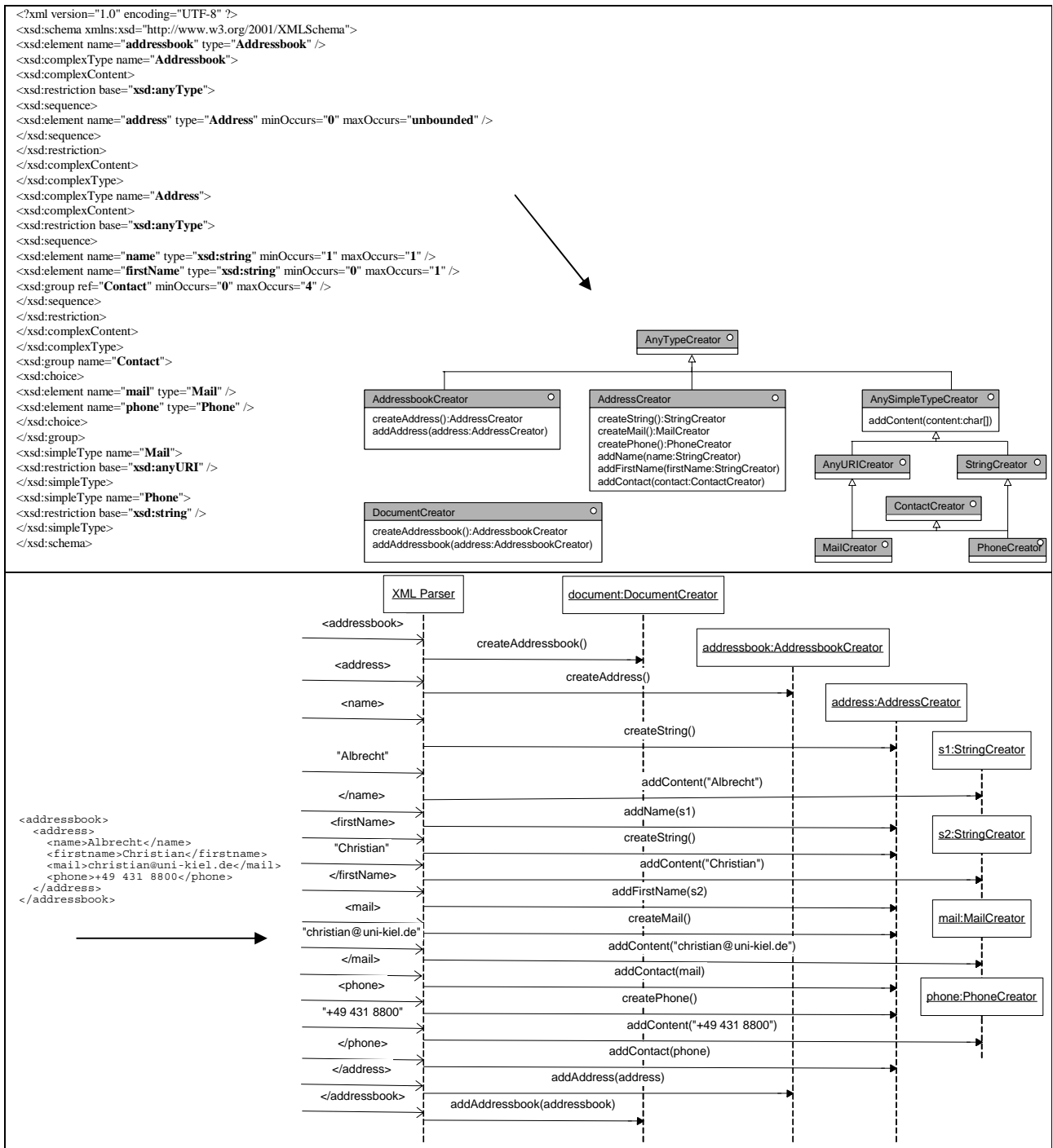
which leads to the following interface structure by applying rule I:



Rule II: Union

The interface structure generated for simple type definitions which define new simple types by uniting a set of other simple types is altered by applying the façade-like pattern in the same way as above.

Figure 5: The address book sample



For every member type of the union a generalization is added and the `createTypeCreator` method in referencing types is replaced by the appropriate `create` methods.

Consider for the example the schema fragment:

```
complexType name="Root">
<element name="value" type="IntegerOrUnbounded"/>
```

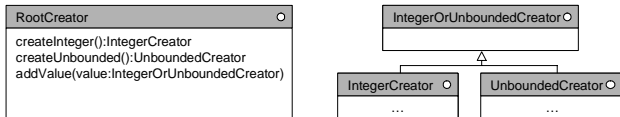
```
</complexType>
<simpleType name="IntegerOrUnbounded">
  <union memberType="xsd:integer unbounded"/>
</simpleType>
<simpleType name="unbounded">
  <restriction base="xsd:token">
```

```

    <enumeration value="unbounded"/>
  </restriction>
</simpleType>

```

By applying rule II the following interface structure is generated:



7. Naming

The interfaces generated by the STAX/bc-binding compiler need to be named. If the type declaration from which an interface should be generated is anonymous, then the binding compiler must generate a name. For code quality this name should be meaningful, i.e. the binding compiler should not generate type names from "unknown0" to "unknown999".

We next present the naming applied in the STAX/bc-binding compiler. The rules are summarized in figure 4.

Rule I: Anonymous type definitions

Anonymous type definitions are named with the surrounding declaration's name. Additionally a number may be appended when conflicts must be resolved.

For example the element declaration which contains an anonymous type definition

```

<element name="a">
  <complexType>
    <sequence>...</sequence>
  </complexType>
</element>

```

is mapped to:

```

<element name="a" type="A">
  <complexType name="A">
    <sequence>...</sequence>
  </complexType>

```

Rule II: Anonymous model group definitions

Anonymous model group definitions are named by creating a new named group whose name is build out of the model group variety (all, sequence or choice) and the concatenation of the particles' names. Again, additionally a number may be appended when conflicts must be resolved.

Rule III: Anonymous simple type definitions without surrounding declarations

Anonymous simple type definitions without surrounding declarations are named by prefixing the **BaseType** with the enumeration values and suffixing the **BaseType** with the given bounding values.

8. STAX/bc Programming

We will now explain the STAX/bc-based programming by example of the address book schema shown in fig. 5. It models an address book which contains a set of address entries. Each address entry contains a name, optionally a first name and up to four contacts. A contact information is either an e-mail address or a telephone number. A sample address book is shown in fig. 5.

The interfaces generated by the STAX/bc-compiler (structural improvements enabled) are also shown in fig. 5. For convenience we also illustrated in the sequence diagram of fig. 5 how the STAX/bc-runtime system calls the generated interfaces when the address book sample of fig. 5 is parsed.

We will now develop an Java-application which deserializes the sample address book of fig. 5 directly into a Swing instance as shown in fig. 6. This is done without any intermediate representation.



Figure 6.

First we implement the interfaces StringCreator, MailCreator and PhoneCreator as a Swing JLabel:

```

public class MyString extends JLabel implements
StringCreator, PhoneCreator, MailCreator {
  public void addContent(java.lang.String content) {
    setText(content);
  }
}

```

Next we implement the AddressCreator interface as a Swing JPanel which holds the JLabels:

```

public class MyAddress extends JPanel implements
AddressCreator {
  public StringCreator createString() {
    return new MyString();
  }
  public void addName(StringCreator name) {
    add((MyString) name);
  }
  public void addFirstname(StringCreator name) {
    add((MyString) name);
  }
  public ContactCreator createPhone() {
    return new MyString();
  }
  public ContactCreator createMail() {
    return new MyString();
  }
  public void addContact(ContactCreator contact) {
    add((MyString) contact);
  }
}

```

Finally the AddressbookCreator interface is implemented by a Swing JFrame which holds the addresses in a tabbed pane:

```

public class MyAddressbook extends JFrame implements
AddressbookCreator {
  private JTabbedPane pane = new JTabbedPane();
  MyAddressbook() {
    super("STAX sample");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().add(pane);
  }
  public AddressCreator createAddress() {
    return new MyAddress();
  }
  public void addAddress(AddressCreator address) {
    pane.add("Address", (MyAddress) address);
  }
}

```

The following implementation of the DocumentCreator interface will instantiate the frame:

```

public class GUI implements DocumentCreator {
  public AddressbookCreator createAddressbook() {
    return new MyAddressbook();
  }
  public void addAddressbook(AddressbookCreator
addressbook) {
    ((MyAddressbook) addressbook).pack();
    ((MyAddressbook) addressbook).show();
  }
}

```

Further applications of the STAX/bc-API are e.g. the storage of XML encoded data into relational data bases. By performing appropriate INSERT resp. UPDATE SQL-functions within the create and add methods, XML content can easily be stored into a relational data base – typed and on-the-fly.

9. Conclusion

We presented the STAX/bc data binding compiler. STAX/bc is the first binding compiler which generates event-based data binding APIs out of W3C schema descriptions. Therewith STAX/bc unites the advantages of data binding frameworks together with the advantages of event-based XML-APIs. These are:

- the processing of large XML documents and data streams as well as on-the-fly processing
- free choice of data structures used to maintain the data
- easy access to the information within your programming language due to specially tailored interfaces.

STAX/bc generated XML-APIs are successfully deployed in several applications. Many of these applications use non-trivial W3C schemata.

The STAX/bc binding compiler is integrated in the <<astax-Framework which is available in an early alpha version at <http://www.ccastax.net>.

Literature

- [1] Fabio Simeoni, David Lievens, Richard Connor, Paolo Manghi. Language Bindings to XML. IEEE Internet Computing 7(1), Jan., Feb. 2003, p. 19-27.
- [2] Robert van Engelen, Gunjan Gupta, Saurabh Pant. Developing Web Services for C and C++. IEEE Internet Computing 7(2), Mar., Apr. 2003, p. 53-61.
- [3] Joseph Williams. J2EE vs. .NET, The Web Services Debate. Communications of the ACM 46(6), June 2003, p. 59-63.
- [4] Gerry Miller. .NET vs. J2EE. Communications of the ACM 46(6), June 2003, p. 64-67.
- [5] Haruo Hosoya, Benjamin C. Pierce. XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology (TOIT) 3(2), May 2003.
- [6] Richard Connor, David Lievens, Fabio Simeoni, Steve Neely, George Russell. Projector: A Partially Typed Language for Querying XML. Plan-X: Programming Language Technologies for XML, 2002.
- [7] Paolo Manghi, Fabio Simeoni, David Lievens, Richard Connor. Hybrid Applications over XML: Integrating the Procedural and Declarative Approaches. Proceedings of the fourth international workshop on Web information and data management 2002, McLean, Virginia, USA. Nov. 2002.
- [8] Simeoni, F., Manghi, P., Lievens, D., Connor, R.C.H. and Neely, S. An approach to high-level language bindings to XML. Information and Software Technology 44(4), Mar. 2002, p. 217-228.
- [9] Brett McLaughlin. Java & XML Data Binding. O'Reilly & Associates. 1st. ed. 2002.
- [10] Hiroshi Maruyama, Kent Tamura, Naohiko Uramoto, Makoto Murata, Andy Clark, Yuichi Nakamura, Ryo Neyama, Kazuya Kosaka, Satoshi Hada. XML and Java: developing Web applications. Pearson Education. 2nd ed. 2002.
- [11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: elements of reusable object-oriented software. Addison-Wesley Publishing. 1996.
- [12] Eric van der Vlist. XML Schema. O'Reilly & Associates. 1st. ed. 2002.
- [13] Dan Suciu. The XML Typechecking Problem. SIGMOD Record 31(1), Mar. 2002, p. 89-96.
- [14] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. Proceedings of the ACM International Conference on Functional Programming, 2003.
- [15] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML Schema Languages using Formal Language Theory. Extreme Markup Languages 2000, August 13-14, 2000. Montreal, Canada.
- [16] XML Data Binding Resources. <http://www.rpbouret.com/xml/XMLDataBinding.htm>.
- [17] Code Fast, Run Fast with XML Data Binding. <http://java.sun.com/xml/jaxp/dist/1.0.1/docs/binding/DataBinding.html>.
- [18] XML and Java technologies: Data binding. <http://www-106.ibm.com/developerworks/xml/library/x-databdopt/>.
- [19] Java Specification Request 31: XML Data Binding Specification. <http://www.jcp.org/en/jsr/detail?id=031>.
- [20] Java Architecture for XML Binding (JAXB). <http://developer.java.sun.com/developer/technicalArticles/WebServices/jaxb/>.
- [21] Castor. <http://www.castor.org>.
- [22] jbind. <http://jbind.sourceforge.net/>.
- [23] Zeus. <http://zeus.enhydra.org/>.
- [24] Quick. <http://qare.sourceforge.net/web/2001-12/products/index.html#quick>.
- [25] SAX. <http://www.saxproject.org>.
- [26] DOM. <http://www.w3.org/DOM/>.
- [27] JDOM. <http://www.jdom.org/>.
- [28] XML Schema Definition Tool (Xsd.exe). <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconxmlschemadefinitiontoolxsdx.exe.asp>.
- [29] XML Information Set. <http://www.w3.org/TR/xml-infoset/>.
- [30] XML Processing Plus Plus. A typed and stream-based XML processing extension for Java. <http://alphaworks.ibm.com/aw.nsf/FAQs/xmlprocessingplusplus>