COMPLETE AND REUSABLE DESCRIPTION OF MESSAGE STRUCTURAL CONSTRAINTS IN WEB SERVICE INTERFACES[1]

Ales Frece [a] (ales.frece@fri.uni-lj.si), Matjaz B. Juric [a] (matjaz.juric@fri.uni-lj.si)

[a] University of Ljubljana, Faculty of Computer and Information Science,
Trzaska cesta 25, SI-1000 Ljubljana, Slovenia

Abstract

Existing specifications for describing message structure as a part of web service description do not support use case-specific definition of structural constraints. We propose a solution to describe a complete set of structural constraints for a particular business object in all its use cases. To implement our solution we use XML Schema (XSD), de facto standard for description of web service message structure. We propose XSD extensions that realize two distinct and complementary approaches. Measurements have shown that by using our extensions the average complexity of real world schemas (XSD documents) comparing to expressional equivalent alternatives is smaller by ~29%.

Keywords

Service description, XML Schema (XSD), use case-specific structural constraints, reuse

1 Introduction

Web service description (i.e. interface) should include constraints on message structure [1] using XSD (XML Schema [2]) [3]. XSD is used for description and validation of structure of XML (Extensible Markup Language [4]) messages being exchanged between web services, WS-BPEL (Web Services Business Process Execution Language [5]) business processes and other service consumers. It is widely observed [6][7][8][9] that XSD fails to achieve the objective to provide useful level of structural constraint definition. Useful level is described as the ability to describe all structural constraints while maximizing reuse of the description parts. This is due to a lack of capabilities for use case-specific definition of elements and types that need to be used in different use cases [10][11], especially regarding the definition of obligatory elements. Solution proposed by the problem observers is (A) to specify elements with specific constraints regarding optionality in different use cases as optional. Such best practice is recommended by standardization organizations, such as NIST [6] and TM Forum [8], advisory organizations, such as Fiatech [7], and industry content providers, such as IBM [9]. In this best practice approach (A), presence of obligatory elements in messages is not validated by the XSD, as these constraints vary for different use cases [6]. This approach has several negative consequences on development and maintenance of final software solution (described in detail in Section 3). However, it is argued by [6][7][8][9] that all these drawbacks weigh out the ones of the obvious alternative solution that proposes (B) usage of a different XSD element/type for each specific use case containing exact structural constraints for that use case. This is mainly due to a significantly poorer reuse [7] (discussed in detail in Section 3).

We believe that all the described disadvantages of the approach (A) by comparison with the approach (B) can be eliminated by enabling use case-specific definitions of structural constraints on XSD types and

[1] This article appeared in: Computer Standards & Interfaces 35 (2013) 218–230.

elements for different use cases. This way all described disadvantages of the approach (B) can be avoided and all described disadvantages of the approach (A) eliminated.

The main objective of this article is therefore to propose a solution to enable use case-specific definitions in XSD. Using our solution obligatory elements are defined in a single element/type for all its use cases, thus number of XSD types in a domain is reduced to a minimum. Consequentially service reuse is the same as in the approach (A). As argued by proponents of the approach (A) it is better than in the approach (B). XSD definitions are used for complete structure validation in all use cases, including checking obligatory elements. There is no additional effort needed for development and maintenance of business logic for this kind of checking. Service descriptions using our solution are self-documented because all structural constraints are defined in XSD. Service consumers can determine the set of obligatory data for each particular use case from the service description alone. Thus, service discovery process is neither aggravated nor prolonged.

We achieve our main objective by proposing XSD extensions for use case-specific definition of XSD types and elements that are used in different use cases, in each with its own set of specific constraints. Proposed extensions provide capabilities for use case-specific definitions of structural constraints, most notably to specify specific elements of a complex type to be used only in some use cases. Proposed extensions realize two distinct approaches. Element centric approach enables definition of specific constraints directly on the relevant elements. Use case centric approach enables definition of constraints overrides from the default use case in separate sections.

This article is organized in eight sections. We present related work in Section 2. In Section 3, we present problems our solution solves from a broad perspective and explain why existing solutions are not satisfactory. We describe the problems in detail in Section 4 and the proposed solution in Section 5. In Section 6, we present proof of concept. We discuss the results in Section 0 and give conclusions in Section 8.

2 Related work

Review of the related work has shown that a solution similar to ours does not exist. Coen, Marinelli and Vitali [12] have proposed SchemaPath, a solution to support of co-occurrence constraints in XSD. These constraints describe that if a specific attribute or element is present, another attribute or element must also be (or must not be) present inside the same type. This is a different kind of conditional definition of types as our use case-specific definition. Our solution enables definition of XSD types to be used in different use cases while their solution allows usage of element interdependent conditions. However, with introduction of a control element that would uniquely identify a use case of the type being defined, their solution could be used to realize same goals as ours. Nevertheless, a problem of additional business irrelevant data (control element) in business object (BO) instances emerges when using this approach. This is not a problem in our solution because there is no need for a control element. Additionally, which use cases are available would not be evident in their solution since there is no mechanism to list them.

W3C is working towards the completion of XSD 1.1 [13] that will be mostly compatible with XSD [2], fix its bugs and make some improvements. Among the improvements is introduction of co-occurrence constraints in a similar way as in [12]. By using XSD 1.1 assertions instead of extensions described in [12] similar results with similar drawbacks as described above can be achieved. XSD 1.1 type alternatives implement similar mechanism than our solution but with an important difference. Type alternatives describe different types that can be chosen while our solution describes specific use cases of a single type. This allows our extensions to maximize reuse while XSD 1.1 cannot achieve this objective.

Wang and Liu [14] extended XSD with nonmonotonic inheritance. Their solution supports overriding of inherited elements and attributes, blocking the inheritance, and conflict handling. Solution addresses a

similar overriding mechanism as our use case centric approach. Their extensions alter semantic meaning of some of the XSD elements and attributes. This introduces unnecessary and undesirable ambiguity to schemas (service descriptions are not self-documented). Our solution uses XSD provided extensibility instead. It leaves the original XSD elements and attributes semantics intact. Our solution requires exactly one XSD type for each BO in all its use cases to be defined while their solution requires one type for each of the use cases. Schemas using their approach are therefore more complex and reduce reuse.

W3C recommendation Semantic Annotations for WSDL and XSD [15] introduces a set of extension attributes for WSDL and XSD that allow description of additional semantics of service descriptions. This solution does not specify a language for representing the semantic models (e.g. ontologies) but provides mechanisms by which concepts from the semantic models that are typically defined outside the WSDL and schemas (like in e.g. [16]) can be referenced from within the WSDL and the XSD components using annotations. When expressed in formal languages (the solution is agnostic to semantic representation languages) these semantics can help disambiguate the description of web services during automatic discovery and composition of web services. Using this solution (despite it is not its primary purpose) it would be possible to address the same problems our solution does. However, by using this solution use case-specific constraints of an XSD type would be defined in external documents using a semantic representation language other than XSD. This means that compared to our solution there would be more artifacts to maintain. Furthermore, service consumers would need to understand an additional language for expressing use case-specific constraints. This is not necessary when using our solution.

3 Motivation

SOA (Service Oriented Architecture) encourages reuse and sets increased reuse as one of its goals for a quality integration of systems through web services [17]. Organizations are developing standards to define relevant information units that will be shared (especially guidelines to use XSD capabilities in different contexts) [18]. Web services often supplement input to produce value added output modeled by BOs. A BO describes and models something that belongs to an organization and hence, has a meaning in the business world [19]. Data (contained in a BO instance) is represented in an XML document (e.g. web service message) through hierarchy of elements [20]. For example, invoice BO contained in a message (BO instance) sent to a billing web service holds items to be billed (BO fields represented by XML elements). Web service calculates amount to be paid and taxes (among other). It returns a complete invoice BO instance to be sent to a customer, i.e. received message (billed items) supplemented with calculated data (amount and taxes). Similarly, business processes (e.g. implemented in WS-BPEL [5]) alter, supplement and/or remove specific data in each step. For example, a business process could be implemented to do the billing calculation in several steps by calling different services (one for calculating the amount and the other for calculating the taxes, for example). This inevitably means that in both scenarios there exists one specific XSD type (describing one specific BO) to be used for validation that cannot have static constraints on its structure, namely which elements are obligatory (which are present and which are not).

De facto best practices [6][7][8][9] advise against using different XSD types/elements with fixed structural constraints for specifying a single BO in different use cases (i.e. against the approach (B)). This is due to a significantly poorer reuse of services on the account of a bigger complexity of schemas (XSD documents), mainly on increased number of types and the derived consequences [7] (e.g. the need to implement additional transformations between very similar types). Poorer reuse is in quarrel with one of the most important principles of SOA [17]. The best practices therefore promote using a single XSD type/element that has all (problematic) elements set as optional instead (i.e. promote the approach (A)). However, this approach opens new problems. Part of validation is done through means of XSD (general document structure) and the rest (including obligatory elements in particular use cases) through some other means (usually by business logic). This is a serious drawback of XSD to be used for XML validation

in real world scenarios, such as supplementing data by calling web services (e.g. request contains data for calculation and the service returns that input data with included calculated results in the response). This approach also implies additional development and maintenance effort (manual coding of constraints, additional documenting of these constraints) and an increased possibility for introduction of errors into the final solution (a part of structural constraints is implemented in XSD, the remaining part in business logic). As a consequence XSD also fails to achieve the objective to be self-documented [2]: service consumer has to search for a part of structural constraints in different use cases (e.g. list of obligatory elements for the service to work properly in the desired use case) outside the service description (e.g. in documentation). This aggravates and prolongs service discovery process.

Our approach prevents opening these problems by removing the necessity to set (problematic) elements as optional. Added value of our approach reflects in enablement of a unified solution for describing a single BO including all structural constraints (including obligatory elements) for all the use cases with a single XML type/element, particularly in the following scenarios (and more):

- *Services for data manipulation*: Data in a SOA environment is dispersed through different systems integrated by the architecture. Each system responsible for a specific set of data (specific BOs) usually provides at least services for storing, reading, searching and updating the data (BOs). BO in each of these use cases contains different set of fields. For example, when storing a BO instance it does not yet contain the unique identification (ID) because it is generated when the BO instance is actually stored. When retrieving the BO instance it contains the ID. When searching, as a search result only a specific subset of fields of the BO is returned due to practical reasons (many potential hits). The whole BO instance is retrieved through read operation if needed. When modifying a BO instance specific subsets of BO's fields can be modified independently. When a BO is being modified (at least) its unique identification should be described as obligatory and (at most) all other fields as optional. All these use cases deal with the same BO. Our solution enables description of all structural constraints of this BO in all the use cases with the same XSD type/element.
- *Supplementing and/or removing data in a service*: A request to a service can contain data for calculation. The service returns that input data with included calculated results in the response. In both cases the same BO is used but with different set of obligatory fields. Only one XSD type/element is needed to describe this BO when using our solution.
- *Supplementing and/or removing data in a business process*: Collecting data of a BO in a business process can be accomplished through a series of steps, including calculations, calling services, using human tasks, etc. Business process can also remove certain data from the BO at specific steps. At each step a type/element describing the BO should have a different set of elements defined as obligatory. Our solution enables this kind of type/element definition.

XSD does not provide means to address these kinds of situations [10][11].

4 Problem description

Proponents of the approach (A) [6][7] argue that there may be multiple business transactions that reuse a common piece of schema. Industry proponents of the approach (A) [8][9] provide service descriptions that actually reuse a large part of schemas as a part of their solutions. During our work designing schemas for service description on real world scenarios we encountered several similar design challenges where decision whether to use the approach (A) or the approach (B) was necessary. First three patterns (described in Section 4.1) we identified are also a part of SOA patterns identified in literature [21]. We identified another two patterns described in Sections 4.2 and 4.3.

4.1 Describing BOs of services for data manipulation

SOA solution should provide users with means for data creation [21]. Created data (BO instances) needs to be stored. There is always a system responsible for specific data (BOs). In a SOA solution a service responsible for storing data to a particular system is implemented. Operation for storing specific BO instances requires that BO to be described by an XSD type. The type must describe minimum (all obligatory elements) and maximum (additional optional elements) structure of the BO to store (shown in Listing 1 (a)). When a request for storing a new BO instance is received by a service the service generates ID of the BO instance and stores it. When reading the BO instance from the system, generated ID is included in the BO instance (shown in Listing 1 (b)). These two use cases cannot be specified by the same type in XSD. The problem is difference in the presence of the BO instance's ID attribute. When BO instance is read it always contains its ID. Before it is stored it does not yet contain the ID.

```
<xsd:complexType name="BOstore">
  <xsd:sequence>
    <xsd:element name="srchEl1" ... />
    ...
    <xsd:element name="srchElN" ... />
    <xsd:element name="el1" ... />
    ...
    <xsd:element name="elM" ... />
  </xsd:sequence>
</xsd:complexType>
```

a)

```
<xsd:complexType name="BOread">
  <xsd:sequence>
    <xsd:element name="ID" ... />
    <xsd:element name="srchEl1" ... />
    ...
    <xsd:element name="srchElN" ... />
    <xsd:element name="el1" ... />
    ...
    <xsd:element name="elM" ... />
  </xsd:sequence>
</xsd:complexType>
```

b)

```
<xsd:complexType name="BOupdate">
  <xsd:sequence>
    <xsd:element name="ID" ... />
    <xsd:element name="srch1" minOccurs="0" ... />
    ...
    <xsd:element name="srchN" minOccurs="0" ... />
    <xsd:element name="el1" minOccurs="0" ... />
    ...
    <xsd:element name="elM" minOccurs="0" ... />
  </xsd:sequence>
</xsd:complexType>
```

c)

```
<xsd:complexType name="BOsearchResult">
  <xsd:sequence>
    <xsd:element name="ID" ... />
    <xsd:element name="srchEl1" ... />
    ...
    <xsd:element name="srchElN" ... />
  </xsd:sequence>
</xsd:complexType>
```

d)

Listing 1: Types describing BO to store (a), to read (b), to update (c), and to return as a search result (d)

Stored BO instances need to be managed [21]. This includes updating their stored values. Different sets of stored BO's fields can be modified independently. When a BO instance is being modified along with the data to be updated BO instance's ID must be given (shown in Listing 1 (c)). ID is obligatory while other elements are usually not. Searching stored BO instances is a very common SOA scenario [21]. The BO returned from the system as a search result (shown in Listing 1 (d)) contains only a minor subset of BO's fields. In other words, search results usually contain just a part of each BO instance because of practical reasons (many potential hits can quickly enlarge message size). If needed, the whole BO instance is retrieved through the read operation. These two use cases also cannot be specified by the same type in XSD. This is all the more true for all four described use cases.

4.2 Describing BOs for supplementing and/or removing data in a service

A service request can contain data for calculation. The service returns the input data with included calculated results in the response. The service may remove some input data from the response because it is relevant only for calculation but not for the service consumer.

```
<xsd:complexType name="BOin">
  <xsd:sequence>
    <xsd:element name="el1" ... />
    ...
    <xsd:element name="elN" ... />
    <xsd:element name="in1" ... />
    ...
    <xsd:element name="inM" ... />
  </xsd:sequence>
</xsd:complexType>
```
(a)

```
<xsd:complexType name="BOout">
  <xsd:sequence>
    <xsd:element name="el1" ... />
    ...
    <xsd:element name="elN" ... />
    <xsd:element name="out1" ... />
    ...
    <xsd:element name="outP" ... />
  </xsd:sequence>
</xsd:complexType>
```
(b)

Listing 2: Types describing input BO (a) and output BO as a result of calculation (b)

XSD type specifying the input BO to the service that performs the calculation is shown in Listing 2 (a). XSD type specifying service output BO is shown in Listing 2 (b). These two use cases cannot be specified by the same type in XSD. Using the approach (A) and setting all (problematic) elements as optional would aggravate service discovery and usage. Service consumers would not be able to see from service description alone which elements are obligatory for calling the service and which ones will be certainly returned in the response.

4.3 Describing BOs for supplementing and/or removing data in a business process

Collecting data of a BO in a business process can be accomplished through a series of steps, including calculations, calling services, using human tasks, etc. Business process can also remove certain data at specific steps. At each step a type describing the BO has a different set of elements defined as obligatory (some are added, some removed). For a business process it is very important to validate the data it receives from external sources (services). This is because business processes are central orchestrators or choreographers of web services [17] and therefore responsible for correctness of the results. At the described steps of business processes we encounter the same problem as described in Section 4.2. Unable to use schemas for validation is a huge handicap for business processes. All validation logic for making sure obligatory data is present at each step must be coded manually in the processes. This complicates exception handling, increases business process complexity and is more error prone.

5 Proposed extensions to XSD

XSD does not support definition of use case-specific structural constraints. Our objective is to enable this support. To achieve this objective, we propose specific XSD extensions. Our extensions are designed to support two complementary approaches. The first approach is the *element centric (EC)* approach. It is used to describe use case specific constraints directly on elements, e.g. whether a particular element should be obligatory in a specific use case. The second approach is the *use case centric (UCC)* approach. Its purpose is to define the default use case and the alternative use cases that describe only the overrides from the default use case. Both of the approaches enable overriding of any constraint, including `xsd:minOccurs`, `xsd:maxOccurs` and `xsd:nillable`.

Our extension attributes exploit XSD-provided extensibility mechanism. They are included through a mechanism provided by the `xsd:anyAttribute` definition inside the local and the global `xsd:complexType` element and inside the `xsd:localElement` element. The `xsd:appinfo` element can be used to provide information for tools, stylesheets and other applications [2]. We propose to use it for instructions on how to adapt the default use case into specific alternative use cases in the UCC approach. Inclusion of extension elements into the `xsd:appInfo` element is possible through an extension mechanism provided by its `xsd:any` type definition. This assures backward compatibility with tools that are unaware of the extensions. We present the structure of the extensions through XSD definitions. We present their syntax using Schematron schema [22]. Schematron schema is an XML document describing a set of rules that

invoke expressions in an external query language on XML documents to validate them. Among others supported, XPath 2.0 [23] is preferred. XPath is an expression language that allows processing tree representation of XML documents. By using expressions, Schematron enables definition of non-sequential constraints (among others). Leveraging this feature enables us to formally specify the syntax of our extensions. We explain semantics of our extensions through natural language descriptions.

We use `xsdx` as the namespace prefix for the `http://soa.si/xsd/extensions` namespace, which is the namespace of the proposed extensions. All defined extension types are used inside extension elements and attributes. To name, uniquely identify and reference a use case, we propose a simple type `xsdx:tUseCase` (shown in Listing 3, lines 5-7). It holds the use case name. To list use case names, the `xsdx:tUseCases` list is defined (lines 8-10).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsdx="http://soa.si/xsd/extensions"
3  targetNamespace="http://soa.si/xsd/extensions" elementFormDefault="qualified" attributeFormDefault="unqualified">
4    <xsd:import namespace="http://www.w3.org/2001/XMLSchema" schemaLocation="http://www.w3.org/2001/XMLSchema.xsd"/>
5    <xsd:simpleType name="tUseCase">
6      <xsd:restriction base="xsd:Name"/>
7    </xsd:simpleType>
8    <xsd:simpleType name="tUseCases">
9      <xsd:list itemType="xsdx:tUseCase"/>
10   </xsd:simpleType>
11   <xsd:attribute name="availableUseCases" type="xsdx:tUseCases"/>
12   <xsd:attribute name="usingUseCase" type="xsdx:tUseCase"/>
13   <xsd:attribute name="whenInUseCases" type="xsdx:tUseCases"/>
14   <xsd:attribute name="whenNotInUseCases" type="xsdx:tUseCases"/>
15   <xsd:attribute name="forUseCase" type="xsdx:tUseCase"/>
16   <xsd:element name="adapt" type="xsdx:tAdapt"/>
17   <xsd:complexType name="tAdapt">
18     <xsd:sequence>
19       <xsd:element name="element" type="xsdx:tExtensionElement" maxOccurs="unbounded"/>
20     </xsd:sequence>
21     <xsd:attribute ref="xsdx:forUseCase"/>
22   </xsd:complexType>
23   <xsd:complexType name="tExtensionElement">
24     <xsd:complexContent>
25       <xsd:extension base="xsd:localElement">
26         <xsd:attribute name="doNotUse" type="xsd:boolean" fixed="true"/>
27       </xsd:extension>
28     </xsd:complexContent>
29   </xsd:complexType>
30 </xsd:schema>
```

Listing 3: Proposed extensions

This list is used in the `xsdx:availableUseCases` attribute (line 11). It lists all use cases a specific type can be used in. Therefore it is attached to the top level `xsd:complexType` that can be used in more than one use case. This restriction can be formally described by the first rule in the Schematron pattern in Listing 4 (lines 2-7). The context the rule applies to (the `context` attribute) is the `xsdx:availableUseCases` attribute (line 2). This means that the rule applies to every occurrence of this attribute in the XML document being validated. The `node` variable (declared by the `sch:let` element) references (the `value` attribute) the `xsdx:availableUseCases` attribute's parent (line 3), which is the XML node containing the attribute (i.e. `xsd:complexType`). Assertion (the `sch:assert` element, lines 4-6) tests (the `sch:test` attribute) whether the node is actually the top level (first predicate) XSD `xsd:complexType` (a node with name `complexType` from the XSD `namespace`). If the test fails, a validation error with message contained in the `sch:assert` element text is reported (line 6).

```
1  <sch:pattern name="common">
2    <sch:rule context="@xsdx:availableUseCases">
3      <sch:let name="node" value=".."/>
4      <sch:assert test="count($node/../../xsd:schema) > 0 and local-name($node)='complexType' and namespace-uri($node) =
5  'http://www.w3.org/2001/XMLSchema'">
6        Attribute xsdx:availableUseCases can only be attached to a top level xsd:complexType.</sch:assert>
7    </sch:rule>
8    <sch:rule context="@xsdx:usingUseCase">
9      <sch:let name="node" value=".."/>
10     <sch:assert test="local-name($node)='element' and namespace-uri($node) = 'http://www.w3.org/2001/XMLSchema'">
11       Attribute xsdx:usingUseCase can only be attached to an xsd:element.</sch:assert>
12     <sch:let name="type" value="$node/@type"/>
13     <sch:let name="namespace" value="namespace-uri-for-prefix(substring-before($type,':'), $node)"/>
14     <sch:assert test="some $x in tokenize((doc(/xsd:schema/(xsd:import[@namespace=$namespace]|xsd:include)/@schemaLocation) |
15 /xsd:schema[@targetNamespace=$namespace]/..)/xs:schema/xs:complexType[@name=substring-
16 after($type,':')]/@xsdx:availableUseCases, '\s') satisfies $node/@xsdx:usingUseCase = $x">
17       Attribute xsdx:usingUseCase should reference a valid use case in the xsdx:availableUseCases list of an top level
18 xsd:complexType that is defined in an external (imported, included) or the same document.</sch:assert>
19   </sch:rule>
20 </sch:pattern>
```

Listing 4: Restrictions describing common extension elements syntax

The `xsdx:usingUseCase` attribute (line 12 in Listing 3) is used on an `xsd:element` to name a particular use case that should be used when its type (defined in the `xsd:type` attribute) has more than one use case available. It can be attached to `xsd:element` only. The latter rule is formally described by the first assertion of the second rule shown in Listing 4 (lines 10-11). Secondly, `xsdx:usingUseCase` must name one of the available use cases of the element's type. This is formally defined by the last assertion in the second rule (lines 14-18). In this rule in addition to the `node` variable (line 9) `type` and `namespace` variables are declared (lines 12 and 13 respectively). The `type` variable holds the XSD type of the node (the node is actually an XSD element). The `namespace` variable holds the namespace of the element derived from the `node`'s `type`. The assertion tests whether the use case named in `xsdx:usingUseCase` is actually contained in `xsdx:availableUseCases`. To achieve this, the appropriate (complex) type must be found first. The type is being searched based on the name and namespace. For the search all imported documents with the proper namespace (`xsd:import`), all included documents (`xsd:include`), and the current document (/) are considered. When the proper type is found, its list of available use cases is tokenized (`tokenize()`). The named use case is then compared to all tokens to find if it matches any of the tokens (`some … in … satisfies …`). In a case that the named use case is not found in the tokens, a validation error is reported (lines 17-18).

5.1 Element centric approach extensions

In the EC approach, we define use case specific constraints of an element in a XSD complex type directly on that particular element. The `xsdx:whenInUseCases` attribute (line 13 in Listing 3) is used on `xsd:element` in a type that is used in at least two use cases. The attribute lists all the use cases the wrapping element definition refers to. In contrast, the `xsdx:whenNotInUseCases` list (line 14 in Listing 3) holds the names of all the use cases the wrapping element definition does not refer to (i.e. it refers to all available use cases not contained in the list). These two attributes can be used to define elements with the same name and type but with completely different constraints in different use cases. This way, any aspect of the elements can be overridden in any of the use cases. A formal description of the requirement that the two extension attributes must be used on `xsdx:element` is defined in the `whenUseCases` abstract rule shown in Listing 5 (lines 2-11). Abstract rules are rules without specified context (no `context` attribute, line 2). Rules are declared abstract by specifying the `abstract="true"` attribute (line 2). In our case we use this abstract rule in the context of both of the extension attributes supporting the element centric approach by specifying the

contexts (lines 12 and 15) and referencing the abstract rule through both `sch:extends` elements (lines 13 and 16).

```
1   <sch:pattern name="elementCentric">
2     <sch:rule abstract="true" id="whenUseCases">
3       <sch:let name="node" value=".."/>
4       <sch:assert test="local-name($node)='element' and namespace-uri($node) = 'http://www.w3.org/2001/XMLSchema'">
5         Attribute <sch:name/> can only be attached to an xsd:element.</sch:assert>
6       <sch:let name="this" value="."/>
7       <sch:assert test="count(/xsd:schema/xsd:complexType[every $x in tokenize($this, '\s') satisfies (some $y in
8   tokenize(@xsdx:availableUseCases, '\s') satisfies $x = $y)]//*[deep-equal(.,$node/..)]) > 0">
9         Attribute <sch:name/> should reference a valid use case in the xsdx:availableUseCases list of its parent xsd:complexType (parent
10  is a top-level xsd:complexType that contains the whole xsd:typeDefParticle [group|all|choice|sequence] with the $node)</sch:assert>
11    </sch:rule>
12    <sch:rule context="@xsdx:whenInUseCases">
13      <sch:extends rule="whenUseCases"/>
14    </sch:rule>
15    <sch:rule context="@xsdx:whenNotInUseCases">
16      <sch:extends rule="whenUseCases"/>
17    </sch:rule>
18  </sch:pattern>
```

Listing 5: Restrictions describing element centric extension elements syntax

The abstract rule requires that both of the attributes are attached to `xsd:element` (lines 4-5). It defines the `this` variable (line 6) that holds the relevant attribute (`xsdx:whenInUseCases` or `xsdx:whenNotInUseCases` depending on the context when actually using the abstract rule). The second assertion (lines 7-10) checks if the relevant attribute actually addresses one of the available use cases that selected XSD type offers. To achieve this, the selected XSD type must be found first. The `xsd:complexType` definition is in the same document but can be an undefined number of levels higher in the tree than the `node` is. Therefore its location is determined by searching the `xsd:complexType` definition with the type definition particle (`xsd:group`, `xsd:all`, `xsd:choice` or `xsd:sequence`) containing the `node` (`deep-equal()` and //*). Then, the `xsdx:availableUseCases` attribute of the found type is tokenized and tokens compared against named use cases in a similar way than with the `xsdx:usingUseCase` attribute. The only difference is that in this case many use cases can be listed so every listed use case must be present in the available use cases list (`every … in … satisfies …` added).

5.2 Use case centric approach extensions

In the UCC approach, the default use case is defined first. All alternative use cases are derived from the default use case and describe differences and overrides of the constraints. We propose the `xsdx:adapt` element (line 16 in Listing 3) and the `xsdx:forUseCase` attribute (line 15) to define and to use these alternative use cases. We propose the `xsdx:element` element (lines 19) for definition of overrides of a particular element inside the `xsdx:adapt` element (its type specified in lines 17-22). The element name is used for correlation between the original definition and the adaptation. This is enough since XSD restricts if the particle contains two or more element declarations with the same name and target namespace, then all their type definitions must be the same [2]. If the use case would not override any part of the element's definition, the definition from the default use case would be used. The `xsdx:element` extension element is of the same type as `xsd:element` (line 25), so any constraint on the element definition can be overridden in the `xsdx:adapt` section. All constraints that are not overridden in a particular use case are used from the default one. We propose an additional fixed attribute `xsdx:doNotUse="true"` (line 26) to be used in `xsdx:element` to specify that particular element definition in the default use case should not be used in a specific alternative use case. The `xsdx:forUseCase` attribute is used in a type definition particle (`xsd:group`, `xsd:all`, `xsd:choice` or `xsd:sequence`) in the enclosing `xsd:complexType` to name the default use case. It is also used in the `xsdx:adapt` element to name each individual alternative use case the element describes

(line 21). The value of the attribute should actually reference one of the available use cases that the named XSD type offers. This check can be formally presented by the first rule in Listing 6 (lines 2-9). The rule checks reference integrity by tokenizing the value of the `xsdx:availableUseCases` attribute and comparing the tokens with the value of the `xsdx:forUseCase` attribute in a similar way as the `whenUseCases` rule in Listing 5. The only difference is that the rule for `xsdx:forUseCase` is simplified because the attribute's value is one use case name and does not need to be tokenized.

```
1   <sch:pattern name="useCaseCentric">
2     <sch:rule context="@xsdx:forUseCase">
3       <sch:let name="this" value="."/>
4       <sch:let name="node" value=".."/>
5       <sch:assert test="count(/xsd:schema/xsd:complexType[some $x in tokenize(@xsdx:availableUseCases, '\s') satisfies
6   $this = $x]//*[deep-equal(.,$node)]) > 0">
7         Attribute xsdx:forUseCase should reference a valid use case in the xsdx:availableUseCases list of its parent
8   xsd:complexType (parent is a top-level xsd:complexType that contains the $node)</sch:assert>
9     </sch:rule>
10    <sch:rule context="xsdx:adapt">
11      <sch:let name="node" value=".."/>
12      <sch:assert test="local-name($node)='appinfo' and namespace-uri($node) = 'http://www.w3.org/2001/XMLSchema'">
13        Attribute xsdx:adapt can only be attached to an xsd:appinfo.</sch:assert>
14    </sch:rule>
15  </sch:pattern>
```

Listing 6: Restrictions describing use case centric extension elements syntax

The `xsdx:adapt` element is used to specify overrides in alternative use cases definitions. Each `xsdx:adapt` element describes all overrides for one alternative use case. Alternative use case name is specified in the `xsdx:forUseCase` attribute (line 21 in Listing 3). Each override of an individual element in the default use case is specified in its own `xsdx:element` (line 19) of type `xsdx:tExtensionElement` (lines 23-29). The `xsdx:tExtensionElement` type is an extension (includes additional fixed attribute `xsdx:doNotUse="true"`) of the `xsd:localElement` type. This is the same XSD type used to define elements in the default use case. This way any default use case detail can be overridden. The `xsdx:adapt` element should be included into the `xsd:appinfo` element of the complex type annotation section. This requirement is formally specified by the second rule in Listing 6 (lines 10-14). If an `xsdx:element` element does not reference a valid `xsd:element` within the enclosing `xsd:complexType` it means that the element should be added to the type at the end of the type definition particle (same behavior as with the XSD mechanism provided by the `xsd:extension` element).

6 Proof of concept

To demonstrate problems that our extensions address more clearly, we describe a simplified billing service (an example of use case described in Section 4.2). We use this example to demonstrate how our extensions can be leveraged to address the described problems, how they are properly used according to described syntax, and to make their semantics more tangible. Example billing service receives prepared invoice message containing (among others) items to be billed and ID of the price list to be used for billing. Service then retrieves price list based on provided ID, performs billing and calculates (among others) the total amount to be billed and the tax summary. When service finishes, it returns completed invoice that now includes calculated total amount in the invoice summary and tax summary. Service removes the price list ID from the invoice, because this data is not relevant for further use.

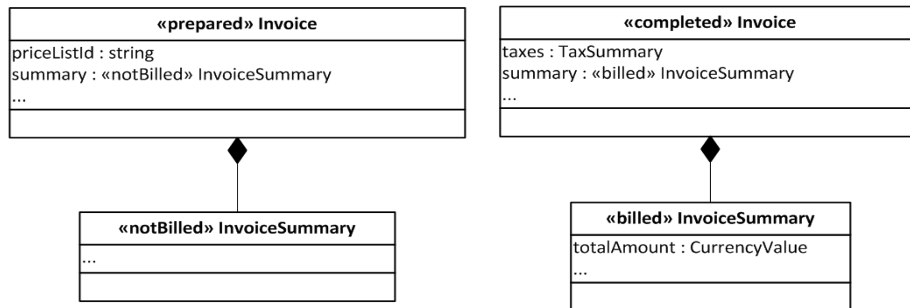

Figure 1: Invoice with two different use cases

Message description of such a billing service includes definition of the invoice BO in XSD. The invoice XSD type is used for message description and validation in two use cases: when the service is called (a request message received by the service) and when the service returns the result (a response message received by a service consumer). Data elements that must be contained in the invoice in these two use cases are different. E.g., when service is called, price list ID is present while the tax summary is not (see `«prepared» Invoice` in Figure 1). When the service returns the result, price list ID is removed and the tax summary is inserted (see `«completed» Invoice` in Figure 1). It is obvious that validation of obligatory elements cannot be achieved with a single XSD type that is used in both use cases. Double type definitions or setting all relevant elements optional is unavoidable. However, we can overcome this problem with our extensions.

6.1 Element centric approach

In Listing 7, we have defined a simple invoice with the `tns:tInvoice` complex type (lines 6-14). In order to use the extensions in a schema, we must import definitions of extensions into the schema through `xsd:import` (line 4). We have imported an external schema (line 5) that contains complex types used in the example, such as `external:tTaxSummary` (line 9) and `external:tCurrencyValue` (line 17).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://example.si/invoice" xmlns:xsdx="http://soa.si/xsd/extensions"
3  xmlns:external="http://external.si" targetNamespace="http://example.si/invoice" elementFormDefault="qualified" attributeFormDefault="unqualified">
4    <xsd:import namespace="http://soa.si/xsd/extensions" schemaLocation="extensions.xsd"/>
5    <xsd:import namespace="http://external.si" schemaLocation="example_external.xsd"/>
6    <xsd:complexType name="tInvoice" xsdx:availableUseCases="prepared completed">
7      <xsd:sequence>
8        <xsd:element name="priceListId" type="xsd:string" xsdx:whenInUseCases="prepared"/>
9        <xsd:element name="taxes" type="external:tTaxSummary" xsdx:whenNotInUseCases="prepared"/>
10       <xsd:element name="summary" type="tns:tInvoiceSummary" xsdx:usingUseCase="notBilled" xsdx:whenInUseCases="prepared"/>
11       <xsd:element name="summary" type="tns:tInvoiceSummary" xsdx:usingUseCase="billed" xsdx:whenNotInUseCases="prepared"/>
12       <!-- items to be billed, company/party info, dates from/to, notes ... -->
13     </xsd:sequence>
14   </xsd:complexType>
15   <xsd:complexType name="tInvoiceSummary" xsdx:availableUseCases="notBilled billed">
16     <xsd:sequence>
17       <xsd:element name="totalAmount" type="external:tCurrencyValue" xsdx:whenInUseCases="billed"/>
18       <!-- already paid amount, due amount, past liabilities ... -->
19     </xsd:sequence>
20   </xsd:complexType>
21   <xsd:element name="preparedInvoice" type="tns:tInvoice" xsdx:usingUseCase="prepared"/>
22   <xsd:element name="completedInvoice" type="tns:tInvoice" xsdx:usingUseCase="completed"/>
23 </xsd:schema>
```

Listing 7: Example using extensions in the element centric approach

With `xsdx:availableUseCases` on `tns:tInvoice`, we have listed all the use cases the invoice can be used in (line 6). We have defined one element per each of the `tns:tInvoice` use cases (lines 21 and 22). We have used the `xsdx:usingUseCase` attribute to name the appropriate use case. When the billing web service is called, the `prepared` use case is used to validate the data (line 21). When the billing web service returns the response, the `completed` use case is used (line 22). The invoice is structured as follows. Price list ID is present only in the `prepared` use case and not present in the `completed` use case. We have described this situation by using the `xsdx:whenInUseCases` attribute (line 8). On the other hand, the tax summary is calculated by the web service. So, this element can be marked to be required in all use cases but the `prepared` one with the `xsdx:whenNotInUseCases` attribute (line 9). The invoice summary is a complex type with two available use cases (`notBilled` and `billed`, line 15). It comprises of the total amount (line 17) and other attributes (line 18). When data is prepared for billing, the total amount is not present (the `notBilled` use case) and is filled only later after the billing is complete (`billed` use case). The total amount to be required in the `billed` use case only is achieved by listing the `billed` use case in the `xsdx:whenInUseCases` list (line 17). The invoice summary in the `notBilled` use case is included into the invoice in the `prepared` use case by listing the `notBilled` use case in the `xsdx:usingUseCase` attribute and the `prepared` use case in the `xsdx:whenInUseCases` list (line 10). When the billing completes (the `completed` invoice), the invoice summary with the total amount (the `billed` use case) is present in the response. This is achieved by listing the `billed` use case in the `xsdx:usingUseCase` attribute and the `prepared` use case in the `xsdx:whenNotInUseCases` list (line 11).

```
<xsd:complexType name="BO" availableUseCases="store read update search">
  <xsd:sequence>
    <xsd:element name="ID"                               whenNotInUseCases="store" ... />
    <xsd:element name="srchEl1"     minOccurs="0"        whenInUseCases="update" ... />
    <xsd:element name="srchEl1"                          whenNotInUseCases="update" ... />
    ...
    <xsd:element name="srchElN"     minOccurs="0"        whenInUseCases="update" ... />
    <xsd:element name="srchElN"                          whenNotInUseCases="update" ... />
    <xsd:element name="el1"         minOccurs="0"        whenInUseCases="update" ... />
    <xsd:element name="el1"                              whenInUseCases="store read" ... />
    ...
    <xsd:element name="elM"         minOccurs="0"        whenInUseCases="update" ... />
    <xsd:element name="elM"                              whenInUseCases="store read" ... />
  </xsd:sequence>
</xsd:complexType>
```

Listing 8: Four types from Section 4.1 specified with only one type using the EC approach

In Listing 8, we have defined a common type for all four use cases from Section 4.1 using the EC approach. Extension attributes (bold font) and alternative element definitions (italic font) have been added.

```
<xsd:complexType name="BO" availableUseCases="in out">
  <xsd:sequence>
    <xsd:element name="el1"          ... />
    ...
    <xsd:element name="elN"          ... />
    <xsd:element name="in1"          whenInUseCases="in" ... />
    ...
    <xsd:element name="inM"          whenInUseCases="in" ... />
    <xsd:element name="out1"         whenInUseCases="out" ... />
    ...
    <xsd:element name="outP"         whenInUseCases="out" ... />
  </xsd:sequence>
</xsd:complexType>
```

Listing 9: Two types from Section 4.2 specified with only one type using the EC approach

In Listing 9, we have defined a common type for both use cases from Section 4.2 using the EC approach. Extension attributes (bold font) have been added. Alternative element definitions from other use case (italic font) have been added.

6.2 Use case centric approach

In Listing 10, we have defined a simple invoice from Listing 7 using the UCC approach. Imports and elements derived from the `tns:tInvoice` type are not shown in Listing 10, because they are specified the same way as in the EC approach. Available use cases are also listed the same way.

```
1   <xsd:complexType name="tInvoice" xsdx:availableUseCases="prepared completed">
2     <xsd:annotation>
3       <xsd:appinfo>
4         <xsdx:adapt xsdx:forUseCase="completed">
5           <xsdx:element name="priceListId" doNotUse="true"/>
6           <xsdx:element name="taxes" type="external:tTaxSummary"/>
7           <xsdx:element name="summary" type="tns:tInvoiceSummary" xsdx:usingUseCase="billed"/>
8         </xsdx:adapt>
9         <!-- other possible use cases ... -->
10      </xsd:appinfo>
11    </xsd:annotation>
12    <xsd:sequence xsdx:forUseCase="prepared">
13      <xsd:element name="priceListId" type="xsd:string"/>
14      <xsd:element name="summary" type="tns:tInvoiceSummary" xsdx:usingUseCase="notBilled"/>
15      <!-- items to be billed, company/party info, dates from/to, notes ... -->
16    </xsd:sequence>
17  </xsd:complexType>
18  <xsd:complexType name="tInvoiceSummary" xsdx:availableUseCases="notBilled billed">
19    <xsd:annotation>
20      <xsd:appinfo>
21        <xsdx:adapt xsdx:forUseCase="notBilled">
22          <xsdx:element name="totalAmount" doNotUse="true"/>
23        </xsdx:adapt>
24        <!-- other possible use cases ... -->
25      </xsd:appinfo>
26    </xsd:annotation>
27    <xsd:sequence xsdx:forUseCase="billed">
28      <xsd:element name="totalAmount" type="external:tCurrencyValue"/>
28      <!-- already paid amount, due amount, past liabilities ... -->
30    </xsd:sequence>
31  </xsd:complexType>
```

Listing 10: Part of the example using extensions in the use case centric approach

The default use case is defined inside `xsd:sequence` for `tns:tInvoice` (lines 12-16) and `tns:tInvoiceSummary` (lines 27-30). The default use case name is specified in the `xsdx:forUseCase` attribute (lines 12 and 27). The `xsdx:forUseCase` attribute could be attached not only to `xsd:sequence` but also to other `xsd:typeDefParticle` group members, including the `xsd:all`, `xsd:choice` and `xsd:group` elements. Alternative use cases are defined in the type's `xsd:annotation/xsd:appinfo` section (lines 3-10 for `tns:tInvoice` and lines 20-25 for `tns:tInvoiceSummary`). The `xsdx:adapt` element is used to adapt particular elements inside the default use case to reflect one of the alternative use cases (lines 4-8 and 21-23). The `xsdx:forUseCase` attribute holds the name of the reflected use case (lines 4 and 21). For example, the default `prepared` use case of `tns:tInvoice` specifies the price list ID element to be obligatory (line 13). However, the `completed` use case overrides the `priceListId` definition and states it should not be used by using the fixed attribute `doNotUse="true"` (line 5). The `completed` use case of the `tInvoice` type has an additional element `taxes` (line 6). The alternative use case `billed` of the `summary` element is specified (line 7) instead of the `notBilled` use case (line 14).

```
<xsd:complexType name="BO" availableUseCases="store read update search">
  <xsd:annotation>
    <xsd:appinfo>
      <xsdx:adapt xsdx:forUseCase="store">
        <xsdx:element name="ID" doNotUse="true"/>
      </xsdx:adapt>
      <xsdx:adapt xsdx:forUseCase="update">
        <xsdx:element name="srchEl1" minOccurs="0"/>
        ...
        <xsdx:element name="srchElN" minOccurs="0"/>
        <xsdx:element name="el1" minOccurs="0"/>
        ...
        <xsdx:element name="elM" minOccurs="0"/>
      </xsdx:adapt>
      <xsdx:adapt xsdx:forUseCase="search">
        <xsdx:element name="srchEl1" minOccurs="0"/>
        ...
        <xsdx:element name="srchElN" minOccurs="0"/>
        <xsdx:element name="el1" doNotUse="true"/>
        ...
        <xsdx:element name="elM" doNotUse="true"/>
      </xsdx:adapt>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:sequence xsdx:forUseCase="read">
    <xsd:element name="ID" ... />
    <xsd:element name="srchEl1" ... />
    ...
    <xsd:element name="srchElN" ... />
    <xsd:element name="el1" ... />
    ...
    <xsd:element name="elM" ... />
  </xsd:sequence>

</xsd:complexType>
```

Listing 11: Four types from Section 4.1 specified with only one type using the UCC approach

In Listing 11, we have defined a common type for all four use cases from Section 4.1 using the UCC approach. Please notice that in the `xsdx:adapt` sections on certain elements only the `xsd:minOccurs` constraint is overridden, so all other constraints from the default use case are preserved. Certain elements are not used (specified by the `doNotUse="true"` attribute). Correlation between the default definition of an element and the adaptation is done by the names of the elements.

```
<xsd:complexType name="BO" availableUseCases="in out">
  <xsd:annotation>
    <xsd:appinfo>
      <xsdx:adapt xsdx:forUseCase="out">
        <xsdx:element name="in1" doNotUse="true"/>
        ...
        <xsdx:element name="inM" doNotUse="true"/>
        <xsdx:element name="out1" ... />
        ...
        <xsdx:element name="outP" ... />
      </xsdx:adapt>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:sequence xsdx:forUseCase="in">
    <xsd:element name="el1" ... />
    ...
    <xsd:element name="elN" ... />
    <xsd:element name="in1" ... />
    ...
    <xsd:element name="inM" ... />
  </xsd:sequence>
</xsd:complexType>
```

Listing 12: Both types from Section 4.2 specified with only one type using the UCC approach

In Listing 12, we have defined a common type for both use cases from Section 4.2 using the UCC approach. Please notice that in the `xsdx:adapt` section certain elements are not used and that correlation between the default definition of an element and the adaptation is done by the names of the elements.

6.3 Schema complexity measurement

With our extensions all use cases of a BO are specified with a single XSD type. This way number of types in a SOA solution is reduced to a minimum. An alternative solution using the same number of types is the approach (A). This approach uses pure XSD schemas with all elements in the types as optional. Disadvantages of this approach are described in Section 3. Another alternative to our extensions not having the disadvantages of the approach (A) is the approach (B). This approach promotes specifying different use cases of a single BO with different XSD types. This solution uses enlarged number of types and consequentially it has several very important disadvantages (as described in Section 3). Since we address complete description of service message structural constraints, we compare complexities of the both alternatives that describe complete structure of BOs in all use cases. These are the approach (B) and the approach using our extensions. Any advantage of our extensions in a form of lower complexity of schemas speaks in favor of our solution to present a better foundation for a bigger reuse (according to promoters of the approach (A)). Especially important is to minimize the number of types needed in a specific domain.

Complexity of schemas can be measured with several different metrics [24][25]. To compare complexities of schemas describing complete structures of BOs in all use cases with and without the proposed extensions, we identified the three most appropriate metrics:

- Number of complex types metric that counts complex types necessary to specify BOs in a specific domain with all their use cases.
- Number of local elements metric that counts elements contained directly inside the complex types.
- Number of lines metric that counts lines in a tree-style formatted schemas from a specific domain, where elements hierarchy is reflected in new line spaces (like in Listing 3).

Others metrics [24][25] not relevant to our objectives are binary file size (we are not optimizing file size), number of XML nodes (we are not dealing with tree representations of schemas), number of

global/local/all element / simple type / complex type / attribute / model group / attribute group definitions (we are dealing with BOs described by complex types and fields of BOs described by elements only), code-oriented breadth/depth (it is highly dependent on XSD mechanisms but is unaware of our adaptation mechanisms, therefore it would not give relevant results), instance-oriented breadth/depth (we are not dealing with XML documents derived from the schemas), and McCabe's complexity (we are not dealing with algorithms that need to "intelligently walk" the tree representation of schemas).

We have a collection of schemas from different projects, mainly carried out in the telecommunication and the power distribution areas. We have chosen one set of schemas from each of these two industries. We have measured complexities of all schemas inside a particular set. We have realized these schemas with our extensions and compared their complexity with the original ones. Results are gathered in Table 1. Totally almost 130 BOs described using the approach (B) with almost 240 complex types (covering all use cases of BOs) and over 700 local elements were considered.

Table 1: Comparison of schema complexity using the approach (B) and using our extensions

Metric	*Electro distribution client lifecycle (EC appr.)*	*Electro distribution client lifecycle (UCC appr.)*	*Telecom customer orders (EC appr.)*	*Telecom customer orders (UCC appr.)*	*Average [b] (by metric)*
Number of complex types	-52%	-52%	-39%	-39%	-46%
Number of local elements	-28%	-20% [a]	-22%	-8,1% [a]	-20%
Number of lines	-26%	-13%	-28%	-16%	-21%
Average [b] (by approach/schema set)	-35%	-28%	-30%	-21%	-29% [b]

[a] Values include all `xsdx:element` elements inside `xsdx:adapt` elements.

[b] Based on rounded values. Data from individual metric/approach/schema set used in calculation is not weighted.

Usage of our extensions positively affects all of the metrics in both sets. The number of complex types has been reduced for almost a half regardless of the selected approach. This is because by using our extensions every BO is specified as a single complex type. All complex types describing alternative use cases become obsolete. Consequentially, number of local elements and lines is reduced for around a fifth. On average, complexity of the schemas is reduced by almost 30%. We have achieved better overall results with the Electro distribution client lifecycle set of schemas than with the Telecom customer orders set. This is due to larger number of use cases per one BO in the first set (as shown by the number of complex type metric).

There is a clear advantage of our extensions in a form of lower complexity of schemas. We have significantly minimized the number of types needed in a specific domain. According to promoters of the approach (A), this way we have fulfilled the necessary condition to increase reuse. By enabling support to describe each BO in all its use cases with a single complex type, we have fulfilled this condition as much as possible (wherever a BO is needed, its sole schema description is used). However, this could be questioned by an increase in the number of local elements or lines (the unified types would be more complex). Fortunately, the number of local elements and lines also decreased (no more double definitions of the same elements in similar types). All these findings speak in favor of our solution to present a better foundation for a bigger reuse of schemas (comparing to the approach (B)) while enabling definitions of all structural constraints within the service descriptions (in contrast to the approach (A)).

7 Discussion

Schemas using our extensions specify exactly which elements are obligatory in which use cases while the approach (A) does not. This way we have eliminated all disadvantages of the approach (A). XSD enhanced with our extensions can provide full data validation. Therefore there is no more need for a separate validation of obligatory elements outside the XSD. This eliminates additional effort for development and maintenance of separate validation rules and the possibility to introduce errors in these rules. Service descriptions using our extensions become self-documented since all structural constraints are captured within the schemas. Therefore service discovery process is eased and shortened (e.g. checking additional documentation is not necessary). In contrast to the approach (B), schemas using our extensions define only one XSD type instead of many to cover all use cases of a particular BO. By achieving that, we have avoided all disadvantages of the approach (B). The most important benefit is bigger reuse. Schema complexity is reduced (one complex type for all possible use cases of a particular BO). The need to implement additional transformations between very similar types describing one BO is avoided. These benefits can be achieved by using either of the proposed approaches (EC or UCC). However, the approaches differ and each of them has its own advantages and disadvantages that mainly relate to effort needed to develop, maintain and use the extended schemas.

The EC and the UCC approaches are complementary. The EC approach provides better readability when we are interested in constraints of one particular element through different use cases. On the other hand, UCC approach provides better view on the constraints on all elements in particular use cases. Comprehension complexity of use cases is very similar to the one for derived types in XSD. When we have relatively complex types with just a few differences, the EC approach is a better choice. The UCC approach better suits situations when we have many differences between relatively simple types. When we have complex use cases, it is much easier to add or remove particular use case when selecting the UCC approach. When we expect changes in specific element's constraints, the EC approach is more suitable for making changes on a single element through different use cases. UCC approach is more convenient for making changes to multiple elements inside a particular use case. Selecting the most suitable approach for problem given depends on the problem itself and on evaluation of expected changes range to emerge in the future. Proper structuring of XSD types eases development of extended schemas and simplifies introduction of changes. Nevertheless, we have summed up the comparison of the EC and the UCC approaches in Table 2 regarding the most common characteristics and situations of schema design. Namely, we have addressed readability, usability and change introduction such as adding, removing and editing different schema particles. By demonstrating advantages and disadvantages of both of the approaches, Table 2 can help schema designers to decide when to use the EC and when the UCC approach. We have grayed the approach that has a better particular characteristic or is more appropriate for usage in a particular situation.

Table 2: Comparison of the element centric and the use case centric approaches

Characteristic / situation		*Element centric (EC) approach*	*Use case centric (UCC) approach*
Readability …	… of a particular element through different use cases	Need to check `xsdx:when(Not)InUseCases` lists *directly* inside the element.	Need to check definition of elements in the complex type and *combine* it with overrides in its annotation section.
	… of a type in a specific use case when we know the type in the default use case	Need to check *each* particular element with extensions if overrides apply to specific use case or not.	Need to check all overrides within *one* `xsdx:adapt` section.
Usability …	… for fewer differences between use cases (simple definitions of types)	*A few elements* that differ from use case to use case. They are *relatively close together*.	All overrides are *gathered in one place*, but *in a different section* as the elements they refer to.
	… for fewer differences between use cases (complex definitions of types)	*A few elements* that differ from use case to use case. They are *dispersed throughout* the complex type.	All overrides are *gathered in one place*, but the elements they refer to are *dispersed* throughout the *other place* in the schema.
	… for many differences between use cases (simple definitions of types)	*Many elements* that differ from use case to use case. They are *relatively close together*.	All overrides are *gathered in one place* within the annotation section of the complex type.
	… for many differences between use cases (complex definitions of types)	*Many elements* that differ from use case to use case are *dispersed throughout* the complex type.	Many complex overrides *gathered in one place*, but the elements they refer to are *dispersed* all over the *other place*.
Adding / removing …	… a simple use case	*Simple changes* in `xsdx:availableUseCases` and in *a few* attributes of relevant elements.	*Simple changes* in `xsdx:availableUseCases` and in the corresponding `xsdx:adapt` section.
	… a complex use case	*Simple change* in `xsdx:availableUseCases` and in *many* attributes of relevant elements.	*Simple change* in `xsdx:availableUseCases` and in the corresponding `xsdx:adapt` section.
Adding / removing / editing …	… one element in one use case	*Change* of *one* element's attributes in *one* place within the complex type.	*Change* of *one* element's overrides in *one* `xsdx:adapt` section.
	… one element in many use cases	*Changes* of *one* element's attributes in *one* place within the complex type.	*Changes* of *one* element's overrides in *many* `xsdx:adapt` sections.
	… many elements in one use case	*Change* of *many* elements' attributes *in many* places within the complex type.	*Change* of *many* elements' overrides in *one* `xsdx:adapt` section.
	… many elements in many use cases	*Changes* of *many* elements' attributes *in many* places within the complex type.	*Changes* of *many* elements' overrides in *many* `xsdx:adapt` sections.

Schemas extended with our extensions are XSD valid. This is important in several aspects, such as development of XSLT (Extensible Stylesheet Language Transformations [26]) stylesheets that transform extended schemas into pure XSD schemas for individual use cases. These auto-generated schemas can be used as an intermediate step for validation of XML documents in particular use cases. This kind of schema pre-processing will also be used in XML Schema Definition Language 1.1 (XSD 1.1) [13] to preserve backward compatibility with XSD. Auto-generated schemas can also be used in cases extended schemas are not suitable, such as in WSDL documents (Web Services Description Language [27]) describing WS-I [28] compliant web services. We see immediate continuation of our work in developing these XSLT stylesheets that would enable schema pre-processing compatible with the one provided by XSD 1.1.

8 Conclusion

We have proposed a solution for complete self-documented description of web service message structure that enables reuse of parts of the description. We implemented our solution by proposing extensions to XSD. Proposed extensions enable definition of XSD types and elements to be used in different use cases, each with its own set of specific structural constraints. We have introduced support for use case-specific definition of XSD types and elements that did not exist before. We have shown that such definition is

important when specific constraints apply only in certain use cases and that existing solutions do not address this problem. Extensions that we have proposed enable redefinition of any constraint on XSD types and elements for particular use cases. With our extensions all use cases of a single BO are specified with a single XSD type. Therefore, the number of types in a SOA solution is reduced to a minimum, which maximizes reuse. Comparing to alternatives, service descriptions using our extensions are self-documented. Service consumers can determine which data is required for a service to work properly in a desired use case from the service description alone. Service discovery process is therefore eased and shortened. Schemas can be used for complete XML structure use case-specific validation, including a set of obligatory elements. Additional effort for development and maintenance of business logic for this kind of checking in not needed. Our extensions consist of two complementary approaches, which we have demonstrated on an example schema. The element centric approach provides better readability of particular elements through different use cases. The use case centric approach provides better view of the complete set of use cases. Our proposed extensions are compatible with the core XSD specification. We have shown that with our extensions the complexity of schemas can be reduced. On two sets of schemas from real-world projects, we have shown that the number of complex types has been reduced by more than 50%. The number of elements inside complex types and the number of lines has been reduced by up to 28%. The average complexity of the schemas has decreased by ~29%. Our extensions not only solve the problem of use case-specific definitions of XSD types and elements, but also reduce the complexity of schemas comparing to alternative approaches.

References

[1] W3C, Web Services Architecture, 2004, http://www.w3.org/TR/ws-arch/, last access: 2.12.2011.

[2] W3C, XML Schema, 2001, http://www.w3.org/XML/Schema, last access: 2.12.2011.

[3] W3C, Web Services Description Language (WSDL) Version 2.0, 2007, http://www.w3.org/TR/wsdl20/, last access: 2.12.2011.

[4] W3C, Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008, http://www.w3.org/TR/xml/, last access: 2.12.2011.

[5] M.B. Juric, WSDL and BPEL Extensions for Event Driven Architecture, Information and Software Technology 52/10 (2010) 1023-1043.

[6] B. Kulvatunyou, K.C. Morris, XML Schema Design Quality Test Requirement Document, NIST - National Institute of Standards and Technology, 2004.

[7] M. Palmer, M. Burns, T. Teague, XML Schema Development Guidelines, Fiatech, 2004.

[8] TM Forum, Information Framework (SID) release 9.5, 2011, http://www.tmforum.org/DocumentCenter/10365/home.html, last access: 2.12.2011.

[9] IBM, WebSphere Industry Content Packs, 2011, http://www-01.ibm.com/software/integration/wicp/, last access: 2.12.2011.

[10] D. Lee, W.W. Chu, Comparative Analysis of Six XML Schema Languages, ACM SIGMOD Record 29/3 (2000) 76-87.

[11] M.S. Ansari, N. Zahid, K.G. Doh, A Comparative Analysis of XML Schema Languages, Database Theory and Application 64 (2009) 41-48.

[12] C. Coen, P. Marinelli, F. Vitali, SchemaPath – A Minimal Extension to XML Schema for Conditional Constraints, in: S. I. Feldman, M. Uretsky, M. Najork, C.E. Wills (Eds.), Thirteenth International World Wide Web Conference, ACM Press, New York, 2004, pp. 164-174.

[13] W3C, XML Schema Definition Language (XSD) 1.1 (Candidate Recommendation), 2011, http://www.w3.org/TR/xmlschema11-1/, last access: 2.12.2011.

[14] G. Wang, M. Liu, Extending XML Schema with Nonmonotonic Inheritance, Lecture Notes in Computer Science 2814 (2003) 402-407.

[15] W3C, SAWSDL - Semantic Annotations for WSDL and XML Schema, 2007, http://www.w3.org/TR/sawsdl, last access: 2.12.2011.

[16] A.-M. Sourouni, G. Kourlimpinis, S. Mouzakitis, D. Askounis, Towards the Government Transformation: An Ontology-based Government Knowledge Repository, Computer Standards & Interfaces 32/1-2 (2010) 44-53.

[17] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall, Upper Saddle River, 2005.

[18] K.C. Morris, A Framework for XML Schema Naming and Design Rules Development Tools, Computer Standards & Interfaces 32/4 (2010) 179-184.

[19] Z. Maamar, R. Charpentier, A Business Object-oriented Environment for CCISs Interoperability, Information and Software Technology 42/3 (2000) 211-221.

[20] B. Carminati, E. Ferrari, Confidentiality Enforcement for XML Outsourced Data, Lecture Notes in Computer Science, 4254 (2006) 234-249.

[21] J. Ganci, A. Acharya, J. Adams, P.D. de Eusebio, G. Rahi, D. Strachan, K. Utsumi, N. Washio, Patterns: SOA Foundation Service Creation Scenario, IBM, 2006.

[22] ISO/IEC, ISO/IEC 19757-2: Information Technology – Document Schema Definition Languages (DSDL) – Part 3: Rule-based Validation – Schematron (Second Ed.), 2008, http://standards.iso.org/ ittf/PubliclyAvailableStandards/c040833_ISO_IEC_19757-3_2006(E).zip, last access: 2.12.2011.

[23] W3C, XML Path Language (XPath) 2.0, 2010, http://www.w3.org/TR/xpath20/, last access: 2.12.2011.

[24] J. Visser, Structure Metrics for XML Schema, in: P. Gomes, J. Machado, J. C. Lopes, J. C. Ramalho, P. Henriques (Eds.), Proceedings of XML: Applications and Associated Technologies, Portland, Texas, 2006, pp. 1-10.

[25] R. Lammel, S. Kitsis, D. Remy, Analysis of XML Schema Usage, in: Conference Proceedings XML 2005, Microsoft, Atlanta, Georgia, pp. 1-38.

[26] W3C, XSLT - XSL Transformations Version 2.0, 2007, http://www.w3.org/TR/xslt20/, last access: 2.12.2011.

[27] W3C, WSDL - Web Services Description Language Version 2.0, 2007, http://www.w3.org/TR/wsdl20/, last access: 2.12.2011.

[28] WS-I - Web Services Interoperability Organization, WS-I Deliverables, 2011, http://www.ws-i.org/deliverables/Default.aspx, last access: 2.12.2011.

Vitae

Ales Frece, B.Sc., is a researcher preparing his doctoral dissertation. His research is focused on business process management and service oriented architecture. He participates in several research and applicative projects, such as business process consolidation and optimization, development of proof-of-concepts, pilots and blueprints. He co-authored WS-BPEL 2.0 for SOA Composite Applications with IBM WebSphere 7 book. He is an IBM SOA Solution Designer and SOA Associate.

Matjaz B. Juric, Ph.D., is Full Professor at the University of Ljubljana and the head of SOA and Cloud Computing Competence Centre. He has authored 15 SOA and Java books, such as Business Process Driven SOA using BPMN and BPEL, SOA Approach to Integration, Business Process Execution Language, BPEL Cookbook (award for best SOA book in 2007), etc. Matjaz has been SOA consultant for several large companies. He has contributed to SOA Maturity Model and performance optimization of RMI-IIOP, etc. He is also a member of the BPEL Advisory Board, an Oracle ACE Director and a Java Champion.