
Universitat Jaume I of Castellón, Spain


http://geotec.uji.es


@geotecUJI


https://www.linkedin.com/company/geotec-geospatial-technologies-research-group---universitat-jaume-i

https://www.youtube.com/user/geotecUJI

**Noname manuscript No.**
(will be inserted by the editor)

# Personalised Code Generation from Large Schema Sets for Geospatial Mobile Applications

**Alain Tamayo · Carlos Granell · Laura Díaz · Joaquín Huerta**

Received: date / Accepted: date

**Abstract** XML and XML Schema are used in the geospatial domain for the definition of standards that enhance the interoperability between producers and consumers of spatial data. The size and complexity of these geospatial standards and their associated schemas have been growing with time reaching levels of complexity that make it difficult to build systems based on them in a timely and cost-effective manner. The problem of producing XML processing code based on large schemas has been traditionally solved by using XML data binding generators. Unfortunately, this solution is not always effective when code is generated for resource-constrained devices, such as mobile phones. Large and complex schemas often result in the production of code with a large size and a complicated structure that might not fit the device limitations. In this article we present Instance-based XML data binding, an approach to produce more compact application-specific XML processing code for geospatial applications targeted to mobile devices. The approach tries to reduce the size and complexity of the generated code by using information about how schemas are used by individual applications. Our experimental results suggest a significant simplification of XML

A. Tamayo
Institute of New Imaging Technologies, Universitat Jaume I, Av. Vicent Sos Baynat, SN, 12071, Castellón de la Plana, Spain
Tel.: +34-964729058
Fax: +34-964728730
E-mail: atamayo@uji.es

C. Granell
European Commission, Joint Research Centre, Institute for Environment and Sustainability, Via E. Fermi 2749, I-21027 Ispra, Italy E-mail: carlos.granell@jrc.ec.europa.eu

L. Díaz
Institute of New Imaging Technologies, Universitat Jaume I, Castellón de la Plana, Spain
E-mail: laura.diaz@uji.es

J. Huerta
Institute of New Imaging Technologies, Universitat Jaume I, Castellón de la Plana, Spain
E-mail: huerta@uji.es

Schema sets to the real needs of client applications accompanied by a substantial reduction of size of the generated code.

**Keywords** XML Processing · XML Schema · Code Generator · Mobile Applications · XML Data Binding

## 1 Introduction

Geospatial applications have become commonplace in desktop, server, and web environments. In such varied scenarios, standardization on communication protocols and data encodings is a vehicle to increase interoperability among data providers and consumers. In the geospatial domain, the Open Geospatial Consortium (OGC[1]) has defined a set of web service interfaces, communication protocols, and data encodings to access, discover, process, visualize and exchange geospatial information in a standard way [1]. The success of these set of standards is witnessed by the large number of OGC-based service instances available online [2]. A geospatial web service that exposes an OGC service specification is termed OGC web service (OWS) throughout this paper.

Increased capabilities in mobile devices (e.g. built-in sensors) have impelled the development of geospatial mobile applications. This trend has also been stimulated by the increasing demand from users to run such applications on their mobile phones [3,4]. While geospatial applications such as navigation systems, web mapping services and location based services are more common everyday on mobile devices, mobile applications that interact with OWS services are still a few. In our opinion, though, the complexity of some data encodings makes it difficult and time-consuming to create reliable and efficient OGC-compliant mobile applications [5]. For example, OGC services may serve huge amount of observational data packed in complex data structures in XML format, which is reputed to be extremely verbose in certain scenarios [6]. While this functioning still runs well in desktop and web settings, it causes great performance penalties and poor efficient processing in mobile devices because of their hardware limitations [7,8]. In the previous example, two issues arise: the size of exchanged messages and the complexity of data structures (i.e., schema complexity). The first issue may be addressed by using compression or some binary versions of XML, but the problem of schema complexity still remains once client applications uncompress incoming messages.

This paper focuses on the schema complexity issue in generating code for OWS-compliant mobile applications. Distinct techniques may address this problem. The first alternative is illustrated by code generators [9–12]. Using generators, developers are relieved from the burden of producing communication and XML processing code manually. However, in some scenarios, finding a code generator that meets all the application requirements (performance, size, scalability, etc.) may be very difficult or even impossible. This is precisely the case of building mobile OWS client applications.

[1] `http://www.opengeospatial.org`

A second alternative would be the use of light-weight data formats such as JSON. These formats do not eliminate the process code for data bindings, that is, XML- or JSON-encoded data must be in the end mapped into application objects. However, this step may be different depending on the target client application. For example JavaScript applications may transform JSON streams in application objects almost automatically. The same though does not hold for Java-based applications since JSON and XML must be handled in a similar manner.

A third alternative would be the definition of application-focused subsets of large XML Schema sets in the form of *application profiles*. For instance, in the geospatial domain some profiles have been defined[2] such as Simple Feature Profile and Common CRS Profile. This strategy makes in principle a lot of sense as the complexity of a source XML Schema is reduced to the portions needed for a particular application. From the perspective of rapid development of client applications, however, it presents some disadvantages. The wide array of potential applications that could use a XML Schema would make it difficult if not impossible to define standardized application profiles for each particular case. Standardization moves forward in a low pace, different from the one of user's and market's needs.

In our opinion, the problem of producing XML processing code for mobile devices in an easy and cost-effective way is greatly limiting the adoption of OWS clients in mobile environments. Built on our previous works [5] and [13], in this article we present an approach called *Instance-based XML Data Binding*.This approach is based on the observation that the existence of large schemas does not imply that every single XML element and type definition is required by client applications [14]. That is, a client application is built according to the real requirements rather than supporting the full range of data structures offered by a given service specification, being most of them particularly unnecessary in that client application. Our approach first automatically extract the subset of a group of specification schemas required by a specific mobile client application. Then this subset is processed by a code generator to produce compact application-specific XML processing code that fits the limitations of mobile devices. An implementation of the approach is presented for Java-based mobile applications running on the Google's Android platform [15]. Nevertheless, the approach is essentially platform-independent and can even be applied to other schema languages beyond XML Schema. We validate our implementation through a set of experiments in a case study in which a generated mobile application retrieves air quality sensor data from OWS services.

The remainder of this article is structured as follows. Section 2 presents an introduction to topics related to XML processing, geospatial schemas and mobile computing. In Section 3, related work on the topic is presented. The approach proposed in this article is presented in Section 4. Section 5 provides details about the implementation of the approach for the selected mobile platform and programming language. After this, Section 6 presents experiments showing the usefulness of this solution using real sensor data. Last, conclusion and future work are presented.

[2] `http://www.opengeospatial.org/standards/profile`

## 2 Background

XML is a text format originally designed to meet the challenges of large-scale electronic publishing [16]. It defines a set of rules to encode documents in a machine-readable form. XML has been adopted as the most common form of encoding information exchanged by Web services [17–19]. XML Schema is used to define the structure of information contained in XML documents [20,21]. Each XML Schema file has a root element named *schema*, that contains the definition of the structure of a set XML documents expressed through schema components such as *complex types, simple types, elements, attributes, and element and attribute groups*. An XML document conforming to the structure defined in some schema is said to be *valid* against that schema.

Producing XML processing code manually is recognised to be difficult and error-prone and may lead to code that is hard to modify and maintain [22,23]. An alternative is the use of code generators to generate this code based on the information contained in XML Schemas. This process is commonly known as XML Data Binding. The use of generators brings benefits such as increased productivity, consistent quality throughout all the generated code, higher levels of abstraction as we usually work with an abstract model of the system, and the potential to support different programming languages, frameworks and platforms [10]. The process of mapping XML Schema components to object-oriented programming constructs is not a trivial one. [24] details all the problems associated to carry out this mapping in a way that satisfies all developer needs. This problem is labelled as *X/O impedance mismatch*. In practice, the most common approach is to perform a best-effort mapping, while recognising that some desirable aspects might still be missing from the generated code.

The availability of many XML data binding code generators has provoked that the task of producing XML processing code be taken for granted by schema designers, who frequently assume that independent of the great length of schemas, a working implementation can be built with little effort. Although this is true in some occasions, with the growth in size of schemas in some domains it may not be the case. For example, this is the case when code generated from large schemas must be accommodated in a device with memory or processing limitations such as mobile devices.

XML Schema allows communities to define their own types on top of a set of predefined types. The geospatial community, under the OGC umbrella, has extensively used XML Schema to define data exchange models for their service specifications. Examples of service specifications are *Web Mapping Service* (WMS) that provides a simple HTTP interface for requesting geo-registered map images from one or more distributed geospatial databases [25]; *Web Feature Service* (WFS) that allows a client to retrieve and update geospatial vector data [26]; and *Sensor Observation Service* (SOS) that provides access to observations from sensors in a consistent manner for all sensor systems, including remote, in-situ, fixed and mobile sensors [27]. Examples of data encodings related to the above service specifications are the *Geography Markup Language* (GML), a grammar for expressing geographical features that serves as a modelling language as well as an interchange format [28–30]; and *Observation and Measurements* (O&M) that defines an abstract model and schema encoding for ob-

servations gathered by sensors [31]. Further readings relating OGC service specifications and data encodings applied to varied application domains are well documented [32,33].

## 3 Related work

In general, issues related with having large and complex schemas have been addressed in multiple domains [34–36]. For example, Pichler et al [34] deal with problems of large schemas in schema matching in the business domain. In the context of schema and ontology mapping Rahm [35] states that current matching systems still struggle to deal with large-scale match tasks, in order to achieve both good effectiveness and good efficiency. Villegas and Olivé [36] present an algorithm to extract fragments from large conceptual schemas arguing that the largeness of these schemas makes it difficult for users to get precise knowledge in which they are interested in.

Literature concerning XML processing for mobile devices is mainly focused in two competing requirements: *information compactness* and *processing efficiency* [7]. To achieve compactness compression techniques are used to reduce the size of XML-encoded information [37,38]. In this sense, the use of compressed format such as EXI [39] would drastically reduce the size of exchanged messages [40], which has positive effect on the volume of transmitted data to mobile devices. Indeed, the addition of compressed formats would only require a few changes in terms of importing libraries for handling these formats in our implementation as we comment later. Compression techniques, however, do not alleviate the issue of parsing exchanged messages (in XML or JSON format) into application objects in the client side.

On the other hand, to the best of our knowledge, not much work has been done to deal with XML processing efficiency in mobile devices. A prominent exception in this topic is the work presented in [7], [38] and [41]. These articles are related to the implementation of a middleware platform for mobile devices: the *Fuego mobility middleware* [42], where XML processing has a large impact. The proposed *XML stack* provides a general-purpose XML processing API called *XAS* [7], an XML binary format called *Xebu* [38], already mentioned before, and other APIs such as *Trees-with-references* (RefTrees) and *Random Access XML Store* (RAXS)[41]. The focus of the aforementioned work is much more low-level than our approach as it deals directly with defining new XML processing APIs and exchange formats. In theory, an implementation of our approach can be built by using any of the above XML processing APIs.

Regarding the use of instance files to drive the manipulation of schemas, a key point in our approach, [35] presents a review of different methods that use instance files for ontology matching. In the field of *schema inference*, instance files are used as well to generate adequate schema files that can be used to assess their validity (e.g. [43–45]).

An extensive literature related to geospatial applications for mobile devices is available [46–50], but mostly unrelated to OGC specifications and standards. Exceptions are a few but give an idea of the increasing importance of producing XML processing code for OGC-compliant mobile applications in an easy and cost-effective

way. Some authors have proposed a kind of proxy approach by implementing a portion of the SOS specification to allow mobile devices to act as an intermediary between physical sensors and SOS servers [51] [52]. Another strategy is driven by the adaptation of Representational State Transfer (REST) style together with light-weight data formats (e.g. JSON) as an alternative to the verbosity of XML-based messages. In this context, an OGC Working Group has recently released a GeoServices REST API candidate standard that leverages JSON as default format for exchanged messages[3]. The proposed REST API allows users to retrieve JSON-encoded geospatial data from main OWS services (e.g. WMS, WFS). This approach eases web mapping applications because of the smooth coupling of JavaScript and JSON formats. Similarly, Rouached et al. [53] proposed a REST interfaces combined with JSON data formats for sensor-related OWS services. Although promising solutions they still do not put their emphasis on addressing the creation of processing code from exchanged messages regardless if they are encoded in XML or JSON formats.

Although most of the data binding generators available nowadays are targeted to desktop or server applications, some tools have been developed for mobile devices such as XBinder[4] and CodeSynthesis XSD/e[5], or for building complete web services communication end-points for resource constrained environments, such as gSOAP [11]. All of the tools mentioned before map XML Schema structures to programming language constructs in a straightforward way, which might not be adequate when large schemas sets are used because it may result in code with a large binary size.

## 4 Instance-based XML data binding

*Instance-based data binding* attempts to generate application-specific data binding code that consumes less computational resources. The approach is based on the assumption that an application does not require all of the *artifacts* (e.g. types) specified in the set of *definition files* (e.g. schemas) used to describe data instances of that application. Note that we use general language to describe the main ideas behind this approach to make obvious that it can be applied to other cases besides XML and XML Schema. For example, it can be also applied to JSON and JSON schemas. We used the term *definition file* to refer to any language or convention used to describe the structure of a set of documents or fragments, and the term *artifact* to refer to any relevant piece of information contained in a definition file.

Figure 1 shows the instance-based data binding approach for XML and XML Schema. In this case, definition files are XML Schemas, artifacts are schema components (types, elements, etc.), and input instances are XML files. The approach also assumes that a representative subset of instances (*input instances*) that must be processed by the application is available. By representative we mean that the input instances contain enough information to infer which artifacts of the definition files are needed to describe all of the instances that will be processed by the application during its lifespan. In our experience working with OWS services such a representative

[3] `http://www.opengeospatial.org/standards/requests/89`

[4] `http://www.obj-sys.com/xbinder.shtml`

[5] `http://codesynthesis.com/products/xsde`

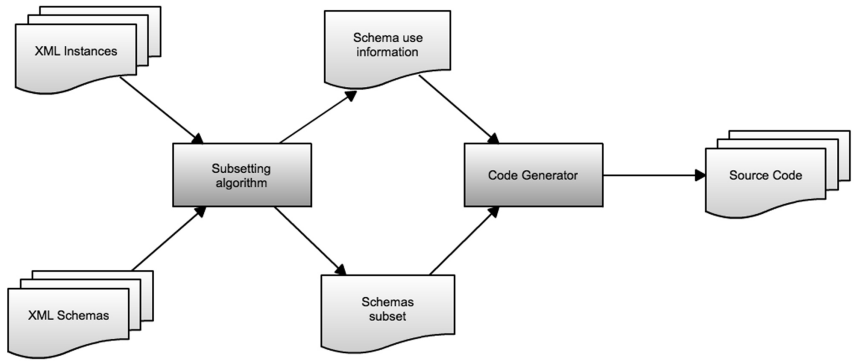


**Fig. 1** Instance-based XML data binding code generation process

subset is frequently available even when the development of a generic client is attempted. In case it were not available synthetic documents that reflect the needs of the application can be built by hand an used as input to the method.

The Instance-based data binding approach consists of two steps. The first step, *instance-based simplification*, extracts the subset of the definition files needed by a given application using information of input instances. The simplification algorithm is used to reduce the number of artifacts in the definition files by removing those that are not necessary to process input instances. The second step, *code generation*, uses information extracted on the previous step to generate data binding code as optimised as possible for a target platform. The "simplified definition files" are not the only input to the code generation process, some other useful information can be extracted from the input instances. The nature of this information is very specific to the type of the definition files, examples for XML and XML Schema will be provided later on this section. Likewise, how this information can be used to produce optimised code will largely depend on the nature of the target platform.

The method outlined above is essentially platform-independent although individual implementations might be targeted to a single platform. In the rest of this section we present a specific implementation of this approach for mobile applications based on OWS standards and targeted on the Android platform. Our main goals in this case are to lower the size of the generated XML data binding code and its memory consumption to cope with the performance limitations of mobile devices. Both goals are mainly accomplished by reducing the number of classes in the generated code, because having a large number of classes have a negative impact on the memory footprint of an application, and in the case of a Java program, in the performance of the class loader [54,55].

To detail some of the features of our solution we will use a specific case study. The case study consists of a mobile client for the Sensor Observation Service (SOS) implementation specification 1.0.0 [27]. The mobile client shall display air quality data of the Valencian Community that available from a 52° North SOS server[6]. This

[6] `http://52north.org/communities/sensorweb/sos/`

sensor data is gathered by a set of control stations located in the Valencian area. The stations measure the level of different contaminants in the atmosphere using a group of sensors. The mobile client application is referred to as *Air quality mobile client* in the remainder of this article. We have chosen SOS because of our previous experience with this specification in the server and client side [5, 14, 40], and also because the SOS schemas are among the most complex schemas of the OGC specifications with more than 700 types distributed over more than 80 schema files. These figures undoubtedly represent a real challenge for testing our approach.

### 4.1 Instance-based schema simplification

The *Instance-based schema simplification* step is aimed to extract the subset of the schemas used on a set of XML documents conforming to them. Starting from the input XML documents the algorithm calculates which schema components are used and which are not. The subsetting algorithm is divided in two steps *XML exploration* and *subset calculation*. During XML exploration the input XML files are processed and the following information is recorded:

- *Schema components that are instanced in XML documents*: For each XML node there exists a global or inner element and a schema type describing its structure. This structure is in turn defined using other schema components. While XML documents are processed the information of the schema components needed to describe the structure each node is recorded.
- *Type and element substitutions*: The subtyping mechanism of XML Schemas allows that the *real* or *dynamic type* of an element in a XML document be different from its *declared type* in the schemas. The information about XML nodes whose dynamic type is different from its declared type is recorded.
- *Wildcard substitutions*: Wildcards are an extensibility feature of XML Schema, which allows to extend the XML document with elements not necessarily specified by the schema. Elements used to substitute wildcards are recorded.
- *Element occurrence constraints information*: for all the elements it is checked that if they allow multiple occurrences there is at least one document where several occurrences of the element are present.

The information of schema components instanced in XML documents is used by the subset calculation step to compute the subset of the XML schemas needed to fully describe the XML input instances. Note that the set of schema components explicitly referenced by instance files is built from other components that are not explicitly referenced. To illustrate how this calculation works we introduce next the concept of *schema set*:

***Definition 1****: An schema set $S = (T_S, E_S, A_S, MG_S, AG_S, R_S)$, where $T_S$ is the set of all type definitions, $E_S$ is the set of all element declarations, $A_S$ is the set of all attribute declarations, $MG_S$ is the set of all element group definitions, $AG_S$ is the set of all attribute group definitions, and $R_S$ is a set of binary relations (described later) between components of $T_S$, $E_S$, $A_S$, $MG_S$, and $AG_S$.*

**Listing 1** XML Schema fragment

```
<complexType  name="Base">
  <sequence>
    <element name="baseElem"  type="string"/>
    <element ref="baseElem2"  minOccurs="0">
  </sequence>
</complexType>

<complexType name="Child">
  <complexContent>
    <extension base="Base">
      <sequence>
        <element name="chdElem"  type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="ContainerType">
  <sequence>
    <element name="item" type="Base" maxOcurrs="unbounded"/>
  </sequence>
</complexType>

<element name="Container" type="ContainerType" />
<element name="baseElem2" type="string" />
```

Components included in sets $T_S$, $MG_S$ and $AG_S$, are composed by inner components. In the case of types, inner components can be references to global elements, attributes, model groups and attribute groups, or they can be nested element and attribute declarations. Model groups can contain references to global elements and other model groups, or they can contain nested element declarations. Similarly, attribute groups may contain references to other global attributes and attribute groups, or they may contain nested attribute declarations. Inner components can be *optional*, meaning that is legal that they do not appear in all valid instance documents. For example in Listing 1, element *baseElem2* in *Base* is optional.

To define the binary relations, we use italics to refer to global types and elements in schema files, e.g. *Container* and *ContainerType*. We refer to attributes or elements within types, model groups or attribute groups, by adding their name and a colon as prefix to the attribute or element name. The whole expression is written in italics, e.g. *ContainerType:item*, *Base:baseElem*, and *Child:chdElem*. So, the binary relations contained in $R_S$ are defined as follows:

- ***isOfType(x, t)***: relates an element or attribute $x$ to its corresponding type $t$. For example: ***isOfType***(*Container*, *ContainerType*), ***isOfType***(*Base:baseElem*, *string*).
- ***reference(x, y)***: relates $x \in T_S \cup MG_S \cup AG_S$ to $y \in E_S \cup A_S \cup MG_S \cup AG_S$ if $x$ references $y$ in its definition using the *ref* attribute in any of its components, e.g. ***reference***(*Base, baseElem2*).

- ***contains(x, y)***: relates $x \in T_S \cup MG_S \cup AG_S$ to $y \in E_S \cup A_S$ if $x$ defines $y$ as an inner attribute or element in its declaration, e.g. ***contains***(*Base*,*Base:baseElem*), ***contains***(*Child*,*child:chdElem*), ***contains***(*Container*, *Container:item*).
- ***isDerivedFrom(t, b)***: relates a type $t$ to its base type $b$, e.g. ***isDerivedFrom***(*Child*, *Base*)
- ***isInSubstitutionGroup(x, y)***: relates an element $x$ to another element y if $y$ is the head element of the x's substitution group.

The schema set S for the schema fragment in Listing 1 remains as follows[7]:

$$\begin{aligned} S = \{&T_S = \{Base, Child, string, ContainerType\},\\ &E_S = \{Container, baseElem2, Base : baseElem,\\ &Child : chdElem, ContainerType : item\},\\ &A_S = \emptyset, MG_S = \emptyset, AG_S = \emptyset,\\ &R_S = \{isOfType = \{(Container, ContainerType),\\ &(baseElem2, string), (Base : baseElem, string),\\ &(Child : chdElem, string),\\ &(ContainerType : item, Base)\},\\ &isDerivedFrom = \{(Child, Base)\},\\ &reference = \{(Base : Base : baseElem2)\},\\ &contains = \{(Base, Base : baseElem),\\ &(Child, Child : chdElem),\\ &(ContainerType, ContainerType : item)\}\\ &isInSubstitutionGroup = \emptyset\} \end{aligned}$$

A schema set can be easily represented as a directed graph where schema components ($T_S$, $E_S$, $A_S$, $MG_S$, $AG_S$, $R_S$) are nodes and binary relations represent edges (Figure 2). Using the information of schema components that are instanced in the XML input instances we can easily find all the components needed to represent these instances by calculating all nodes in the graph that can be reached from the instanced nodes. The initial set of instanced nodes plus all nodes reachable from them represents the output of the subset calculation step. This information is passed to the code generator along with the rest of the information gathered during XML exploration.

### 4.2 Code Generation Process

The second step of *Instance-based data binding* is code generation, and in our case it involves the generation of code for a mobile platform, which poses additional complications when large schema files are used. The constraints related to memory, processing and battery life inhibited that existing generators for desktop or server applications could be easily adapted to these devices. As a consequence, the availability of

[7] XML Schema *anyType* has been omitted purposely to simplify exposition.

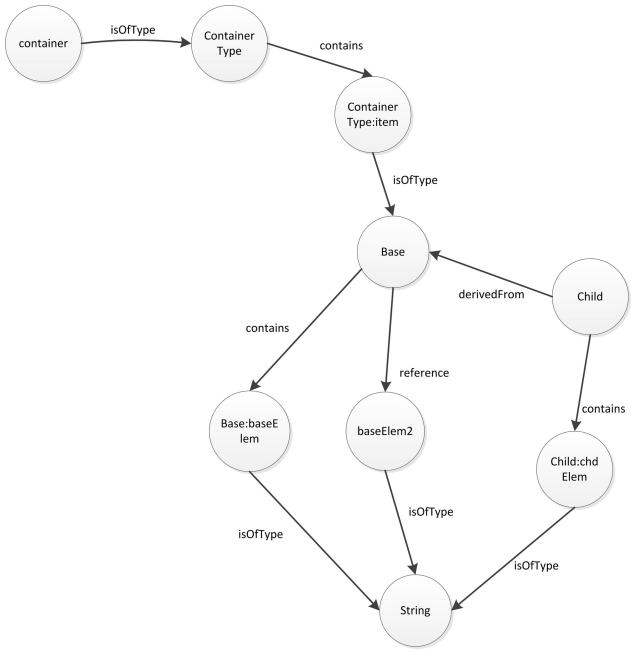


**Fig. 2** Graph of relations in schema fragment in Listing 1

generators for mobile devices is much more limited than for other environments. In our case, we will generate code for the Java programming language and the Android platform[8]. Although a specific language and mobile platform have been selected, in general terms most of the techniques presented here are applicable to any programming language or platform.

A more detailed view of the code generation process is shown in Figure 3. The outputs of the schema simplification step (Section 4.1) are inputs to the *schema processor*, the component of the generator in charge of creating the data model that will be used later by the *template engine*. The *template engine* combines pre-existing *class templates* with the data model to generate the final source code. The use of a template engine allows the generation of code for other platforms and programming languages by just defining new dedicated class templates.

A summary of the features of the code generation process that generates optimised code is listed next:

- *Support for instance-based code generation*: Instance-based code generation refers to the use of information extracted from XML documents to improve the generated code according to some criteria. The use of the information about schema use allows to apply the following optimisations:
  - *Remove unused schema components*: The schema components that are not used are not considered for code generation.

[8] These choices have been made based on the availability of mature tools to implement a prototype which will be used to build a full-fledged sample application.

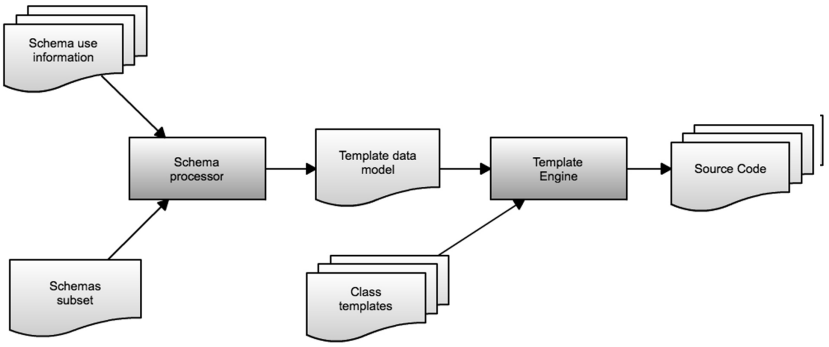


**Fig. 3** Flow diagram for the code generation process

  - *Efficient handling of subtyping and wildcards*: The scenarios involving dynamic typing, i.e, the dynamic type of a node differs from its declared type can be simplified using information gathered from XML documents.
  - *Inheritance flattening*: By flattening subtyping hierarchies it is possible to reduce the number of classes in the generated code.
  - *Adjust occurrence constraints*: An element may be declared to have multiple occurrences but in practice it may have at most one occurrence.
- *Collapse elements containing single child elements*: Information items that will always contain single elements can be replaced directly by its content.
- *Disabling parsing/serialization operations as needed*: Both operations are not always needed. Hence, the generated code can be reduced by not including unneeded operations.
- *Ignoring sections of XML documents*: Frequently, we are not interested in all of the information contained in XML files, offering a good optimisation opportunity by ignoring the unneeded portions.

A detailed explanation of the features related to instance-based code generation is presented in the following subsections. More information about the rest of the features can be found in [13].

#### *4.2.1 Remove unused schema components*

The schema components that are not used are not considered for code generation. By removing the unused components we can substantially reduce the size of the generated code. The simplification algorithm presented in Section 4.1 is in charge of finding the schema components that are used in a set of XML documents, the rest are just ignored during the code generation step.

The amount of the reduction that can be accomplished will depend on how specific applications make use of the original schemas. For example, in [14] a study of the use of XML in a group of 56 servers implementing the SOS specification revealed that only 29.2% of the SOS schemas were used in a large collection of XML documents gathered from those servers. Similarly, [56] showed that less than 10% of the

components of the same schemas were used in the implementation of the *Air quality mobile client*.

### *4.2.2 Efficient handling of subtyping and wildcards*

The complexity of the subtyping mechanisms of XML Schema has been pointed out by several authors [57,58]. These authors stated that this complexity might be the cause why in practice the extra expressiveness of XML Schema is only used to a very limited extent [57]; and that the inclusion of more than one subtyping mechanism might be due to the presence of conflicting design approaches in the *W3C XML Schema Working Group* [58].

As a consequence of this complexity, handling subtyping (and wildcards) can be a complicated issue when code with specific performance requirements must be built, specially if large schemas that rely heavily on these mechanisms are used, such as those included in OGC specifications. In the general case, when no instance-based information is available generic code to face any possible type or element substitution must be written. Nevertheless, this scenario can be substantially simplified with instance-based information. Consider for example the case of complex type *gml:FeaturePropertyType* in the context of the SOS schemas (Listing 2). This type is used as a container for any feature and contains a reference to a global element *gml:_Feature*. *gml:_Feature* is the head element of the substitution group shown in Figure 4, hence, the source code for *gml:FeaturePropertyType* must be ready to parse any of these elements. If instead of generating code for the full SOS schemas, we consider the subset of the schemas needed for the *Air quality mobile clien*t, the number of elements in the substitution group is reduced from 30 to 5 (Figure 5). Even more, we can determine for every specific type which referenced head elements are substituted by which element in its substitution group because this information was recorded during the XML exploration step (see Section 4.1). Using this information we can figure out that class *FeaturePropertyType* only must be aware of parsing elements of type *FeatureCollection* and *SamplingPoint*, as *om:Observation*, *om:ObservationCollection*, and *sa:samplingFeature* never occurred inside *gml:FeaturePropertyType* elements in the XML input instances.

Similarly we can determine which inner elements of a type may have a dynamic type different from its declared type using the information in the *typeSubstitutions* data structure. In the case that type substitution is used for a type, generic code to handle this situation is generated, but when no type substitution is used we can generate simpler code (see Section 5.1.1).

The same technique is applied to wildcards. During the execution of the instance-based schema simplification algorithm it recognizes which elements are used to substitute wildcards. This information is used by the generator to create the appropriate code to handle valid substitutions.

**Listing 2** Extract of *feature.xsd* containing the definition of *gml:FeaturePropertyType*

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.opengis.net/gml">

  <element name="_Feature" type="gml:AbstractFeatureType"
           abstract="true" substitutionGroup="gml:_GML"/>

  <complexType name="AbstractFeatureType" abstract="true">
    <complexContent>
      <extension base="gml:AbstractGMLType">
        <sequence>
          <element ref="gml:boundedBy" minOccurs="0"/>
          <element ref="gml:location" minOccurs="0"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="FeaturePropertyType">
    <sequence minOccurs="0">
      <element ref="gml:_Feature"/>
    </sequence>
    <attributeGroup ref="gml:AssociationAttributeGroup"/>
  </complexType>
</schema>
```

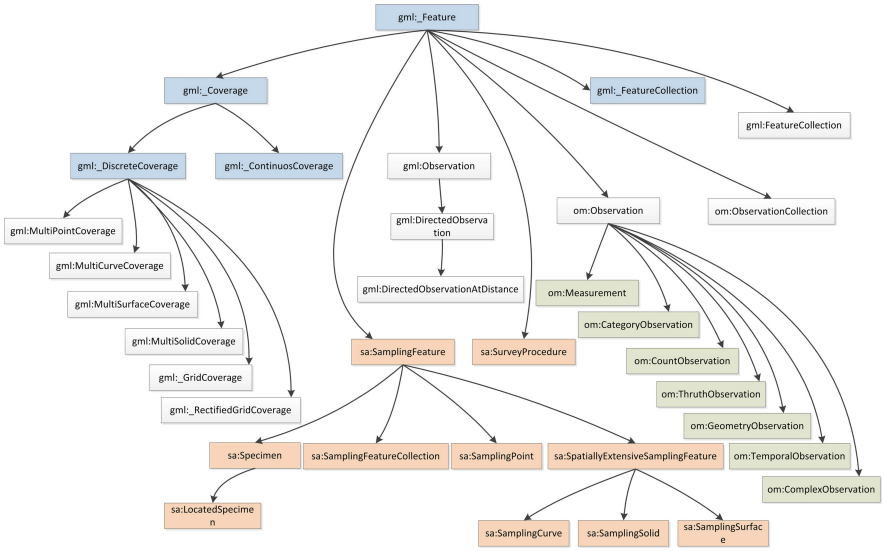


**Fig. 4** Elements in the substitution group of gml:_Feature

### *4.2.3 Inheritance Flattening*

The domain model defined by OWS schemas contains very deep type hierarchies. The value of the *depth of inheritance tree* (DIT)[9] metric for the subtyping hierarchy

[9] DIT is defined as the maximum length from a node to the root of the inheritance tree [59]

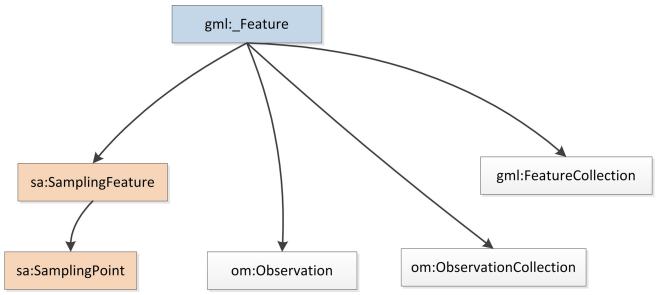


**Fig. 5** Elements in the substitution group of *gml:_Feature* for the air quality mobile client schemas

of GML 3.1.1, an encoding format SOS schemas are built upon, is 7, which is a rather large number. If this specification were analysed in the context of SOS this value would be even bigger as SOS use several GML types as base types for locally defined types.

Type hierarchies in OWS schemas contain many abstract classes, which by definition cannot be instantiated in XML documents. In the code generation step the inheritance tree is "flattened", eliminating all base classes corresponding to types that never need to be instantiated to process input instances. Each class is transformed by adding all of the inherited fields and methods to the class declaration, eliminating its dependency with its base classes. The generated classes will inherit directly from a common base class named *XMLInstanceTag* (see Section 5.1.1).

By flattening subtyping hierarchies we reduce the number of classes in the generated code. The application of this technique will not necessarily result in smaller code as the fields defined in base types must be replicated in all of their child types, but it will have a positive impact in the work of the class loader because a lower number of classes have to be loaded while the application is executed. In the case of geospatial schemas, such as GML with six or more levels, if the hierarchy were not flattened, processing an XML node of a type in the lowest levels of the hierarchy would imply that all its parent types must be loaded first.

Using this technique the generated code is also simpler to understand as a one-to-one correspondence will exist between a schema type and the code in charge of processing a node of this type. Consider the opposite case where the processing is distributed between the class representing the schema type and all its ancestors. Here, the code is harder to understand and even debugging becomes a more complex task.

Inheritance flattening has the advantage that the number of classes on the generated code is reduced, but at the expense of loosing all the information related with subtyping between generated classes, which might not be desirable if the generated code is meant to resemble the domain model and it is not just a mean to access data encoded as XML. This technique has been widely explored and used in different computer science and engineering fields as is proven by the literature found on the topic [60–65]. It is also used to a limited extent in the XML data binding tool JiBX[10].

[10] `http://jibx.sourceforge.net/`

#### *4.2.4 Adjust occurrence constraints*

The *occurrence constraints* of an element determines if it will be mapped to a single object instance or to a list. If the element can have multiple occurrences (*maxOcurrs* $>$ *1* using the XML Schema jargon) it will be mapped to a list, otherwise to a single instance variable. Many elements declared in schemas as having multiple occurrences, have only a single occurrence in XML documents. In this case, they can be safely mapped to a unique instance variable instead of a list, making a better use of memory during the execution of the program. In the schema simplification step the information about element occurrences is stored in the *maxOccurrences* data structure. For a given schema type *T* we can map safely a contained element *E* to a single object instance if a triple *(T,E,1)* is stored in this data structure.

## 5 Implementation

*DBMobileGen* (DBMG for short) is the current implementation of the *Instance-based XML data binding* approach. It includes components implementing both the simplification algorithm and code generation process. It is implemented in Java and relies on existing libraries such as *Eclipse XSD*[11] for processing XML Schemas and *Freemarker*[12] as template engine library. For low-level XML processing we initially used only *kXML*[13] but we have rewritten the code to use any implementation of the XMLPull API[14]. This allows to use implementations of this API based on EXI encodings such as EXIficient[15].

### 5.1 Mapping schema components to programming language constructs

In this section we explain how schemas components are mapped to programming language constructs. The basis of this mapping is very simple, containing rules for mapping complex types, simple types and global elements. To simplify exposition we say that given *s*, a schema component, *T(s)* will be the corresponding programming language construct generated from *s*.

#### *5.1.1 Mapping complex types*

Each complex type in the schemas is mapped to a class in the target programming language (Java in our case), i.e., if $s$ is a complex type, $T(s)$ will be a Java class representing $s$. An UML diagram showing the structure of the class generated for complex type *Child* (Figure 1) is shown in Figure 6. The mapping is performed by applying the following rules:

[11] `http://www.eclipse.org/modeling/mdt/?project=xsd\#xsd`
[12] `http://freemarker.sourceforge.net`
[13] `http://kxml.sourceforge.net/kxml2/`
[14] `http://www.xmlpull.org`
[15] `http://exificient.sourceforge.net`

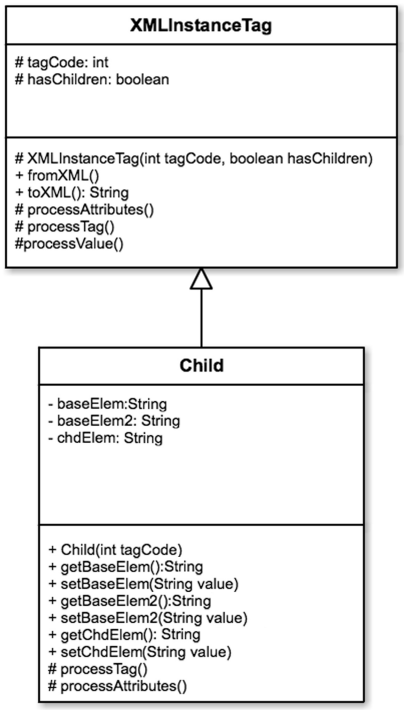


**Fig. 6** Class corresponding to complex type Child

1. Attributes references and declarations are declared as fields of the class. The type of each field, which is always a simple type, is determined according to the rules for mapping simple types.
2. Element references and declarations are declared as fields of the class. The type of each field, which in this case can be a simple or complex type, depends on whether the type of the element on the schemas $t_e$ has a counterpart on the generated code, or is mapped to a primitive Java type. It will also depend on the element occurrence constraints. If the occurrence constraints of an element allow that instances contain at most one occurrence, the type of the element will be $T(t_e)$. If multiple occurrences are accepted it will be mapped to $List < T(t_e) >$.
3. Setter and getter methods are generated for each field.
4. All classes will inherit directly or indirectly from *XMLInstanceTag*, a base class containing the common structure and behaviour of all the mapped classes.
5. An overridden version of the method *processTag* defined in *XMLInstanceTag* is generated for the class. This method will contain the code needed to parse the content of the XML node containing the information to be parsed.
6. A method named *processAttributes* is generated for the class. This method will contain the code needed to parse the attributes of the XML node.

*XMLInstanceTag* is the base class for all types produced during the code generation process. It provides the basic mechanisms to read the content of a node in


**Fig. 7** Class to parse *Capabilities* files

an XML document. The class constructor receives the tag code of the XML node to parse and a boolean value as parameter. The tag code is a unique identifier for each element generated from their names. This code is necessary because Java classes correspond to types in the schemas, and as a consequence elements with different names may have the same type in XML documents. The boolean value defines if the node may have children elements or not. The class also contains the methods *fromXML* and *toXML* for parsing and serialization respectively. *fromXML* reads the content of the current XML node during the parsing process. It first reads the content of attributes, by calling *processAtributtes*, method that must be overridden by classes extending *XMLInstanceTag*. After this, *fromXML* depending whether the node may have children nodes or not, iterates through the nodes processing them as they are reached, or just reads the value of the node. Children nodes are processed inside the method *processTag* that must be overridden by subclasses as well. Similarly, node values are read using the method *processValue*. The method *fromXML* is an implementation of the *Template Method* design pattern [66].

### *5.1.2 Mapping simple types*

Simple types in the schemas will be mapped whenever possible to primitive or predefined Java types. As a general rule, facets that constrain the values of schema predefined types will be ignored to speed up parsing of XML documents. This allows mapping most of the simple types defined in the schemas without having to create new types in the generated code. The only exception will be when a simple type is declared as the union of several types that cannot all be mapped to the same Java type. In that case, a Java class is created with a field for each possible type of the contained values and boolean flags indicating which of the values are set. The approach of not mapping simple types has been selected because the opposite will cause a proliferation of small objects, which consequently will result in the use of more memory and more work for the class loader.

### *5.1.3 Mapping global elements*

By default, global elements are not mapped to any programming language construct unless it is explicitly specified that they can act as roots of XML documents. In that case, a parser class is created with a method to process the instances using a parser implementing the XMLPull API. Figure 7 shows the class in charge of processing a *Capabilities* document containing metadata about the service the mobile client is connecting to.

### 5.2 Limitations

The *Instance-based data binding* approach has an inherent limitation which is that it relies on the existence of a representative set of XML documents that must be processed by the application. This subset might not always be available. In this case, we can still take advantage of the approach by building *synthetic* XML documents containing relevant information. Whether XML processing code is produced manually or automatically developers typically have some knowledge of the structure of the documents that must be processed by the applications. Therefore, we can use this knowledge to build sample XML documents that can be used as input to the simplification algorithm. In case it were necessary, the final code can be later modified manually, or the sample files changed and used to regenerate the code.

The current implementation also presents some limitations. Because of the complexity of the XML Schema language itself, certain features and operations have been only supported and included in DBMG when they were considered necessary for generating client applications [13, 67]. For example, dynamic typing using *xsi:type* is not fully supported and serialization is not supported at all as parsing has more importance in the mobile clients we have used to test the approach.

## 6 Experiments

In this section we use the *Air quality mobile client* introduced earlier to prove how the XML instance-based approach can simplify the development of standards-based geospatial applications. The air quality mobile client must process data retrieved from a given SOS-based server from which we captured a sample of 2492 XML documents to be used, along with the SOS schemas (referred as *full schemas* in the rest this section), as input to our approach. This sample included responses for each operation of the server that will be used for the client. The subset of the schemas (referred as *reduced schemas* in the rest of this section) used by the sample was calculated and stored as XML Schema to be used as input to other generators.

The graph of schema components and their relationships, similar to the one in Figure 4, for the full schemas contains 3333 nodes (schema components) and 4617 arcs (relationships among components). After applying the schema simplification algorithm it was determined than only 327 components and 423 relationships are needed to process the input instances [5], representing only 9.81% and 9.16% respectively of the total number of component and relationships. Special importance among these components have complex types, which are used as primary concepts to generate code, that were reduced from 846 to 112 types (13.23%).

Table 1 shows the size of the code produced by different generators from the full and reduced schemas. Source code generated by DBMG is compiled to *compressed jar* format and compared with final code generated by XBinder, JAXB-RI[16] and XMLBeans[17]. The last two generators are not targeted to mobile devices but are

[16] Java Architecture for XML Binding (JAXB) Reference Implementation: `http://jaxb.java.net/`.

[17] `http://xmlbeans.apache.org/`

**Table 1** Comparing size of code (KBs) for original and simplified schema sets

| | XBinder | JAXB | XMLBeans | DBMG |
|---|---|---|---|---|
| Full | 3,626 | 754 | 2,822 | 88 |
| Reduced | 567 | 90 | 972 | 88 |
| Libs | 100 | 1,056 | 2,684 | 30 |

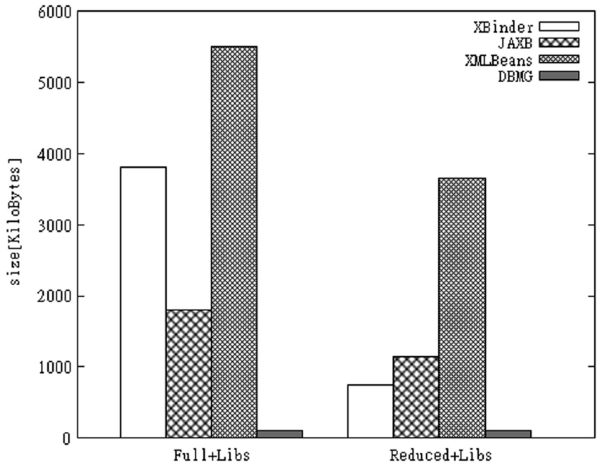


**Fig. 8** Size of generated code for full schemas

used here as reference to compare the size of similar code for other types of applications. The last row of the table (Libs) shows the size of the supporting libraries, which are out of the scope of the simplification algorithm, needed to execute the generated code in each case.

Figure 8 shows the total size of XML processing code when using the full and reduced schemas. In both cases, we can see the enormous difference that exists between the code generated by DBMG and the code generated by other tools. The full and reduced size for DBMG is the same because it implicitly performs the simplification of the schemas before generating source code. It must be noted that serialisation is not still implemented in DBMG whereas the other generators used in the comparison include it. We roughly estimate that including serialisation code in DBMG would increase the final size in about 30%. To calculate this estimate, we manually wrote serialization code for a few cases of varying complexity and measured the increase in size of their compiled files (.class). In all cases we obtained an increase very close to 30%, so we may suggest a similar increase on average because the size of the serialization code depends directly on the numbers of fields contained in every generated class.

From our experiments, the code generated by DBMG is about 6 times smaller than the code generated by XBinder from the reduced schemas and about 30 times

smaller than the code generated from the full schemas. The lack of serialisation support in DBMG partially explains why the code generated by XBinder is a lot bigger. The real factor, however, is that XBinder code ensures all the restrictions related to user-defined simple types and it cannot use instance-based knowledge to simplify the structure of the generated code. Ensuring all simple type restrictions is an advantage if the application requires the data to be carefully validated, but it is a disadvantage in the opposite case, as unneeded verifications increase processor usage and memory footprint. It is also a disadvantage if invalid XML documents must be processed by the target application.

When compared to JAXB, using the reduced schemas, the main difference in size is in the supporting libraries, as the code generated by JAXB is very simple. Still, the code generated by DBMG is slightly smaller because the step of removing elements with single child elements and inheritance flattening eliminates a large number of classes. In all of the cases, XMLBeans has the largest size. This tool is mostly optimised for speed at the expense of generating a more sophisticated and complex code and the use of larger supporting libraries.

At this point an interesting question would be if the results for this specific application can be generalised to other SOS-based or OGC standards-based applications. To try to answer this question, at least partially, we performed in [14] an empirical study, already mentioned in Section 4.2.1, with 56 SOS servers revealing that only 29.2% of the components and 34.45% of the complex types of the SOS schemas were used in a large collection of XML documents gathered from them. Data contained in those servers was related mainly to environmental variables such as *temperature*, *pressure*, *seawater salinity*, and *rainfall*. We used this data as input to DBMG to build a *quasi-generic* SOS client for Android [68]. The size of the generated code for XML processing in compressed jar format was 301 KB.

We have also applied the XML Instance-based approach to build prototypes of geoprocessing mobile applications based on the Web Processing Service (WPS) standard [69]. We built a WPS Explorer that allows users to browse the process descriptions of WPS services and a geoprocessing client based on Google Maps that allows clients to enter simple geometries (points, lines and polygons) used as input to remote processes to execute spatial operations such as *buffer*, *intersection*, and *area*. For the first application only 11 out of the 99 complex types in the WPS schemas were needed. For the second application only 18 out of 395 complex types present in the schemas were necessary to implement the operations mentioned above. The final size of the whole applications in both cases was around 160 KB.

In [13] we tested the performance of the code generated by DBMG which showed to add few overhead to the performance of the underlying parser, kXML, when processed files were below 100 KB. This indicates that the process of creating and initializing application objects do not take a significant amount of time. On the other hand, when file size approaches to 1 MB, the overhead was important ($> 1s$) because the amount of memory required to store the processed information forced the execution of the garbage collector with a higher frequency. We have to keep in mind that code produced manually can have similar problems if it were necessary to retain most of the information read from the XML files in memory. Additionally in [70] performance experiments to compare DBMG to XBinder in a mobile device, and DBMG to

XBinder, JAXB and XMLBeans in two PC configurations and using several datasets were presented. These experiments showed that DBMG outperformed XBinder in most of the test cases but it was slower than JAXB and XMLBeans in PC configurations. This last point is not a surprise as both generators are meant to be used in desktop and server applications, On the other hand, kXML is optimised to be used in resource-constrained devices.

## 7 Conclusion

The approach presented in this article allows the generation of application-specific data binding code with the aim of consuming less computational resources. The approach is based on the assumption that an application does not require all the components used in a set of schemas describing data instances of that application. The approach is thought to be applied not only to XML and XML Schemas but to any other technologies that play a similar role. Parts of the approach can be applied in a platform-neutral way, such as calculating the subset of used components, allowing them to be implemented easily for other platforms or to be combined with existing code generators. Other parts, such as the code generator itself, are platform-specific although the presented implementation is designed to be easily extended to generate source code for different programming languages. Although we present this solution in the context of geospatial mobile applications it can also be extrapolated to other domains for which data specifications exist in some form of schema language.

With this solution we attempt to help developers to control to some extend the complexity of schemas such as those defined by OGC. It can be used as an XML data binding generation tool to build ready-to-use XML processing code or can be used to calculate the subset of the schemas used by an application with the aim of reducing the amount information that must be handled if a manual approach were selected for the implementation of this code. In any case, a substancial simplification of XML Schemas sets to the real needs of client applications is assured.

## References

1. Lee C, Percivall G (2008) Standards-based computing capabilities for distributed geospatial applications. Computer 41:50–57
2. López-Pellicer F, Béjar-Hernández R, Florczyk A, Muro-Medrano P, Zarazaga-Soria F (2011) A review of the implementation of OGC Web Services across Europe. International Journal of Spatial Data Infrastructure Research 6:168–186
3. Anthes G (2011) Invasion of the mobile apps. Commun. ACM 54:16–18
4. Canali C, Colajanni M, Lancellotti R (2009) Performance Evolution of Mobile Web-Based Services. IEEE Internet Computing 13:60–68
5. Tamayo A, Granell C, Huerta J (2011) Dealing with large schema sets in mobile SOS-based applications. In: Proceedings of the 2nd International Conference on Computing for Geospatial Research & Applications, COM.Geo '11, New York, NY, USA. ACM, pp 1–9
6. Barkstrom B (2011) When is it sensible not to use XML?. Earth Science Informatics 4:45–53
7. Kangasharju J, Lindholm T, Tarkoma S (2007) XML Messaging for Mobile Devices: From Requirements to Implementation. Comput. Netw. 51:4634–4654
8. Walker M, Turnbull R, Sim N (2007) Future mobile devices: an overview of emerging device trends, and the impact on future converged services. BT Technology Journal 25:120–125

9. Benatallah B, Casati F, Grigori D, Nezhad H, Toumani F (2005) Developing adapters for web services integration. In: O. Pastor, J. Falcão e Cunha (eds.) Advanced Information Systems Engineering, *Lecture Notes in Computer Science*, vol. 3520. Springer, pp 415–429
10. Herrington J (2003) Code Generation in Action. Manning Publications Co., Greenwich, CT, USA
11. Van Engelen RA, Gallivan KA (2002) The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID '02, Washington, DC, USA. IEEE Computer Society
12. Zimmermann O, Milinski S, Craes M, Oellermann F (2004): Second generation web services-oriented architecture in production in the finance industry. In: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA '04, New York, NY, USA. ACM, pp 283–289
13. Tamayo A, Granell C, Huerta J (2011) Instance-based XML data binding for mobile devices. In: Proceedings of the 3rd International Workshop on Middleware for Pervasive Mobile and Embedded Computing, M-MPAC'2011. Lisbon, Portugal. ACM
14. Tamayo A, Viciano P, Granell C, Huerta J (2011) Empirical study of sensor observation services server instances. In: S. Geertman, W. Reinhardt, F. Toppen (eds.) Advancing Geoinformation Science for a Changing World, *Lecture Notes in Geoinformation and Cartography*. Springer, pp 185–209
15. Google Android.com (2012) `http://www.android.com/`. Accessed 28 Nov 2012
16. W3C (2008) Extensible Markup Language (XML) 1.0 (Fifth Edition). `http://www.w3.org/TR/xml/`. Accessed 28 Nov 2012
17. Kay MH (2003) XML five years on: a review of the achievements so far and the challenges ahead. In: Proceedings of the 2003 ACM symposium on Document engineering, DocEng '03. ACM, pp 29–31
18. Wilde E (2003) XML technologies dissected. IEEE Internet Computing 7:74–78
19. Wilde E, Glushko RJ (2008) XML fever. Commun. ACM 51:40–46
20. W3C (2004) XML Schema Part 1: Structures Second Ed. `http://www.w3.org/TR/xmlschema-1`. Accessed 28 Nov 2012
21. W3C (2005) XML Schema Part 2: Datatypes Second Ed. `http://www.w3.org/TR/xmlschema-2`. Accessed 28 Nov 2012
22. Bray T (2003) XML Is Too Hard For Programmers. `http://www.tbray.org/ongoing/When/200x/2003/03/16/XML-Prog`. Accessed 28 Nov 2012
23. McLaughlin B (2002) Java and XML Data Binding. O'Reilly & Associates, Inc., Sebastopol, CA, USA
24. Lämmel R, Meijer E (2007) Revealing the x/o impedance mismatch: changing lead into gold. In: Proceedings of the 2006 international conference on Datatype-generic programming, SSDGP'06. Springer, pp 285–367
25. OGC (2006) OpenGIS Web Mapping Server Implementation Specification 1.3.0. `http://www.opengeospatial.org/standards/wms`. Accessed 28 Nov 2012
26. OGC (2005) OpenGIS Web Feature Service Implementation Specification 1.1.0. `http://www.opengeospatial.org/standards/wfs`. Accessed 28 Nov 2012
27. OGC (2007) Sensor Observation Service 1.0.0. `http://www.opengeospatial.org/standards/sos`. Accessed 28 Nov 2012
28. Lu CT, Dos Santos R, Sripada L, Kou Y (2007) Advances in GML for Geospatial Applications. GeoInformatica 11:131–157
29. OGC (2004) OpenGIS Geography Markup Language (GML) Implementation Specification 3.1.1. `http://www.opengeospatial.org/standards/gml`. Accessed 28 Nov 2012
30. OGC (2007) OpenGIS Geography Markup Language (GML) Encoding Standard 3.2.1. `http://www.opengeospatial.org/standards/gml`. Accessed 28 Nov 2012
31. OGC (2007) Observations and Measurements - Part 1 - Observation schema. `http://www.opengeospatial.org/standards/om`. Accessed 28 Nov 2012
32. Reichardt M (2010) Open standards-based geoprocessing Web services to support the study and management of hazard and risk. Geomatics, Natural Hazards and Risk 1(2):171–184
33. Foerster T, Schäffer B, Baranski B, Brauner J (2011) Geospatial Web Services for Distributed Processing: Applications and Scenarios. In: P. Zhao, L. Di (ed) Geospatial Web Services: Advances in Information Interoperability. IGI Global, Hershey, pp 245–286
34. Pichler C, Strommer M, Huemer C (2010) Size Matters!? Measuring the Complexity of XML Schema Mapping Models. In: Proceedings of the IEEE Congress on Services. IEEE, pp 497–502
35. Rahm E (2011) Towards large-scale schema and ontology matching. In: Z. Bellahsene, A. Bonifati, E. Rahm (eds.) Schema Matching and Mapping, Data-Centric Systems and Applications. Springer, Berlin, pp 3–27

36. Villegas A, Olivé A (2010) A method for filtering large conceptual schemas. In: Proceedings of the 29th international conference on Conceptual modeling, ER'10. Springer, Berlin, pp 247–260
37. Käbisch S, Peintner D, Heuer J, Kosch H (2010) Efficient and Flexible XML-Based Data-Exchange in Microcontroller-Based Sensor Actor Networks. In: Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops, WAINA '10. pp 508–513
38. Kangasharju J, Tarkoma S, Lindholm T (2005) Xebu: A binary format with schema-based optimizations for XML data. In: Proceedings of the6th International Conference on Web Information Systems Engineering, volume 3806. Springer, Berlin, pp 528–535
39. W3C (2011) Efficient XML Interchange (EXI) Format 1.0. `http://www.w3.org/TR/exi`. Accessed 28 Nov 2012
40. Tamayo A, Granell C, Huerta J (2012) Using SWE Standards for Ubiquitous Environmental Sensing: A Performance Analysis. Sensors 12(9):12026–12051
41. Lindholm T, Kangasharju J (2008) How to edit gigabyte XML files on a mobile phone with XAS, RefTrees, and RAXS. In: Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services, Mobiquitous '08, pp 1–10
42. Tarkoma S, Kangasharju J, Lindholm T, Raatikainen K (2006) Fuego: Experiences with Mobile Data Communication and Synchronization. In: Proceedings of the 2006 IEEE 17th International Symposium onPersonal, Indoor and Mobile Radio Communications, pp 1–5
43. Bex GJ, Neven F, Vansummeren S (2007) Inferring XML schema definitions from XML data. In: Proceedings of the 33rd international conference on Very large data bases, VLDB '07. VLDB Endowment, pp 998–1009
44. Hegewald J, Naumann F, Weis M (2006) XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In: Proceedings of the 22nd International Conference on Data Engineering Workshops. IEEE
45. Min JK, Ahn JY, Chung CW (2003) Efficient extraction of schemas for XML documents. Information Processing Letters 85(1):7–12
46. Doyle J, Bertolotto M, Wilson D (2010) Evaluating the benefits of multimodal interface design for CoMPASS–a mobile gis. GeoInformatica 14:135–162
47. Goh D, Sepoetro L, Qi M, Ramakhrisnan R, Theng YL, Puspitasari F, Lim EP (2007): Mobile tagging and accessibility information sharing using a geospatial digital library. In: D. Goh, T. Cao, I. Slvberg, E. Rasmussen (ed) Asian Digital Libraries. Looking Back 10 Years and Forging New Frontiers, *Lecture Notes in Computer Science*, vol. 4822. Springer, Berlin, pp 287–296
48. Nusser S, Miller L, Clarke K, Goodchild M (2003) Geospatial IT for mobile field data collection. Commun. ACM 46:45–46
49. Simon R, Fröhlich P (2007) A mobile application framework for the geospatial web. In: Proceedings of the 16th international conference on World Wide Web, WWW '07. ACM, , pp 381–390
50. Tsou MH (2004) Integrated mobile gis and wireless internet map servers for environmental monitoring and management. Cartography and Geographic Information Science 31(3):153–165
51. Jändmsä and J, Luimula M, Schulte J, Stasch C, Jirka S, Schöandning J (2010) A mobile data collection framework for the sensor web. In: Proceedings of the Ubiquitous Positioning Indoor Navigation and Location Based Service (UPINLBS). IEEE, pp 1–8
52. Müller R, Fabritius M, Mock M (2011) An OGC compliant sensor observation service for mobile sensors. In: S. Geertman, W. Reinhardt, F. Toppen (Ed) Advancing Geoinformation Science for a Changing World, *Lecture Notes in Geoinformation and Cartography*. Springer, Berlin, pp 163–184
53. Rouached M, Baccar S, Abid M (2012) RESTful Sensor Web Enablement Services for Wireless Sensor Networks. In: Proceedings of the 2012 IEEE 8th World Congress on Services. IEEE, pp 65–72
54. Hartikainen VM, Liimatainen P, Mikkonen T (2006) On mobile java memory consumption. In: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, pp 1–7
55. Wilson S, Kesselman J (2000) Java Platform Performance: Strategies and Tactics. Addison-Wesley, Boston
56. Tamayo A, Granell C, Huerta J (2012) Measuring Complexity in OGC Web Services XML Schemas: Pragmatic Use and Solutions. International Journal of Geographical Information Science 26(6):1109–1130
57. Martens W, Neven F, Schwentick T, Bex GJ (2006) Expressiveness and Complexity of XML Schema. ACM Transactions on Database Systems 31:770–813
58. Møller A, Schwartzbach MI (2006) An Introduction to XML And Web Technologies. Addison-Wesley Longman Publishing, Boston

59. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20:476–493
60. Beyer D, Lewerentz C, Simon F (2000) Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems. In: Proceedings of the 10th International Workshop on New Approaches in Software Measurement. Springer, London, pp 1–17
61. Chirila CB, Ruzsilla M, Crescenzo P, Pescaru D, Ţundrea E (2006) Towards a reengineering tool for java based on reverse inheritance. In: In Proceedings of the 3rd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence
62. Bungartz HJ, Eckhardt W, Mehl M, Weinzierl T (2008) Dastgen–a data structure generator for parallel c++ hpc software. In: Proceedings of the 8th international conference on Computational Science, Part III. Springer, Berlin, pp 213–222
63. Cicchetti A, Ruscio DD, Eramo R, Pierantonio A (2008) Automating co–evolution in model-driven engineering. In: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference, Washington, DC, USA. IEEE, pp 222–231
64. Lagorio G, Servetto M, Zucca E (2009) Flattening versus direct semantics for featherweight jigsaw. In: Proceedsing fo the International Workshop on Foundations of Object Oriented Languages. ACM
65. Bungartz HJ, Eckhardt W, Weinzierl T, Zenger C (2010) A precompiler to reduce the memory footprint of multiscale pde solvers in c++. Future Gener. Comput. Syst. 26:175–182
66. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns. Addison-Wesley, Boston
67. Arago P, Tamayo A, Viciano P, Huerta J, Díaz L (2011) Forest Fire Survey and Processing Tool for Android-Based Mobile Devices. In: Proceedings of the INSPIRE Conference 2011, Edinburgh, Scotland
68. Tamayo A, Viciano P, Granell C, Huerta J (2011) Sensor Observation Service Client for Android Mobile Phones. In: Proceedings of Workshop on Sensor Web Enablement (SWE 2011), Banff, Canada
69. Tamayo A, Granell C, Díaz L, Huerta J (2012) Building Standards-Based Geoprocessing Mobile Clients. In: J. Gensel, D. Josselin, D. Vandenbroucke (eds.) Proceedings of the 15th AGILE International Conference on Geographic Information Science (AGILE 2012), Avignon, France
70. Tamayo A (2011) XML Data Binding for Geospatial Mobile Applications. Phd Thesis, Universitat Jaume I, Castellón de la Plana, Spain `http://www3.uji.es/~atamayo/Phd/Phd_Dissertation-Alain_Tamayo.pdf`. Accessed 19 Mar 2013