Systematic Generation of XML Instances to Test Complex Software Applications[*]

Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini

Istituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche
Via Moruzzi, 1 – 56124 Pisa, Italy
{antonia.bertolino, jinghua.gao, eda.marchetti,
andrea.polini}@isti.cnr.it

Abstract. We introduce the XPT approach for the automated systematic generation of XML instances which conform to a given XML Schema, and its implementation into the proof-of-concept tool TAXI. XPT can be used to automatize the black-box testing of any general application that expects in input the XML instances. We generate a comprehensive set of instances by sampling all the possible combinations of elements within the schema, applying and adapting the well known Category-Partition strategy for functional testing. Originally, XPT has been conceived for application to the e-Learning domain, within which we briefly discuss some examples.

1 Introduction

Increasingly today complex software systems are developed according to a modular architecture, within which precise features can be identified and separately implemented. Main objective of this "componentization" trend is to permit the development of the different features of a complex composite application by diverse stakeholders while maintaining the possibility of integrating the subsystems into a unique working system. Nevertheless the integration clearly presupposes the definition of a rigorous and checkable format of the data exchanged between the components.

One of the most important innovations that strongly contributes to solve this issue has been the introduction of the eXtensible Markup Language (XML) [1]. In few years this language has established itself as the *de facto* standard format for specifying and exchanging data and documents between almost any software application. Immediately following, the XML Schema [2] has then spread up as the notation for formally describing what constitutes an agreed valid XML document within an application domain. Thus, XML Schemas are used for expressing the basic structure of data and parameters that remote components exchange with each other, and restrictions over them, while XML instances, formatted

[*] This work has been supported by the European Project TELCERT (FP6 STREP 507128).

according to the rules of the referred XML Schema, represent the allowed naming and structure of data for components interaction and for service requests.

The introduction of XML for specifying standard format of exchanged data is certainly fundamental and strongly increases the possibility of correct interactions, nevertheless XML related technologies do not solve the interoperability problem per se. No information concerning the interpretation of data can be associated to an XML description, leaving the room for different interpretations by the various developers. Trying to make a further step toward guaranteeing interoperability, our proposal here is to combine the great potential of XML Schema in describing input data in open and standard form, with testing activity to assess the common understanding of interacting e-Learning systems. In doing this, our intention is to take advantage of the special characteristic of the data representation suitable for automated processing, which is clearly a big advantage for testing.

We find that the adoption of the XML Schema leads quite naturally to the application of *partition testing*, a widely studied subject within the testing community, since it provides an accurate representation of the input domain into a format suitable for automated processing. The subdivision of the input domain into subdomains, according to the basic principle of partition testing, can be done automatically by analyzing the XML Schema elements: from the diverse subdomains identified, the application of partition testing amounts to the *systematic derivation* of a set of XML instances. Systematic generation of XML instances, differently from a random based approach, clearly has important consequences on the effectiveness of the generated test suite permitting to derive meaningful statistics on the kind of instances generated, and then on the covered features.

This paper introduces our proposed XML-based Partition Testing (XPT) approach for the systematic generation of XML instances. Also a short overview on a proof-of-concept tool, called TAXI (Testing by Automatically generated XML Instances), is provided. Such tool inputs an XML Schema and automatically generates a set of XML instances for the black box testing of a component, whose expected input conforms to the taken schema. At the same time the paper reports a preliminary qualitative evaluation of the approach to the generation of instances for the IMS Learning Information Package specification.

In the remainder of this paper we discuss some related work in Section 2, and summarise the well known Category Partition method in Section 3; then we provide a description of the proposed strategy in Section 4 and of the tool implementing it in Section 5. Section 6 finally reports some preliminary considerations on application of the methodology; in particular in 6.1 we provide quantitative motivations to the application of a systematic approach, then in 6.2 a simple qualitative comparison of TAXI with another existing tool (XMLSpy) is presented. Some conclusions are finally drawn in Section 7.

2 Related Work

Our research is aimed at automatically generating a comprehensive suite of XML instances from a given XML Schema. The generated XML instances can then be used for the black-box testing of applications that expect such XML instances as input.

Notwithstanding the intense production of XML-based methods and tools in the latest years, to the best of our knowledge there do not exist other XML-based test approaches comparable to ours. Indeed, the existing "test tools" based on XML can be roughly classified under three headlines:

– testing the XML instances themselves;
– testing the XML Schemas themselves;
– testing the XML instances against the XML Schema.

Regarding the first group, a basic test on an XML file instance is *well-formedness*, which aims at verifying that the XML file structure and its elements possess specified characteristics, without which the tested file cannot be even classified as an XML file. Diverse sets of test suites (for instance [3], [4]) and various tools aiming at validating the adequacy of a document instance to a set of established rules, such as [5], have been implemented.

With regard to the testing of XML Schemas themselves (second group), several validators exist for checking the syntax and the structure of the W3C XML Schema document (for instance, [6], [7], [8] [9], and [10]).

The third group encloses tools for automatic instance generation based on XML schema, which is what we also do. Relevant tools of this group are [11], [12] and [13]. However for all of them the XML instances generation only implements random or ad hoc generation; the instances are not conceived so to cover interesting combinations of the schema. Indeed this characteristic is where our approach tries to provide a comprehensive solution. Adopting a systematic criterion in generating instances will have a double positive side effect: the generation of more accurate and mindful XML instances and the automatization of black box test suite specification.

So far no proposal has really succeeded in pushing the widespread adoption of automated black box testing as it would deserve. The well known Category Partition method [14] provides a procedural approach to analyse the input domain and to systematically derive a comprehensive test suite (see a brief description in the next section). It has been previously applied by many authors to requirements specifications expressed in various notations (for instance also by authors of this paper to UML specifications [15]). We propose here to apply it to XML schema. We think in fact that the widespread acceptance of XML, and its pragmatic flavor, associated to the Category Partition methodology could finally be the winning instrument to achieve fully automated black-box testing.

3 Category Partition

Introduced in the late 80's and today widely known and used, the Category Partition (CP) [14] provides a systematic and semi-automated method for test data derivation, starting from analysis of specifications until production of the test scripts, through the following series of steps:

1. Analyze the specifications and identify the *functional units* (for instance, according to design decomposition).
2. For each unit identify the *categories*: these are the environment conditions and the parameters that are relevant for testing purposes.
3. Partition the categories into *choices*:[1] these represent significant values for each category from the tester's viewpoint.
4. Determine *constraints* among choices (either properties or special conditions), to prevent the construction of redundant, not meaningful or even contradictory combinations of choices.
5. Derive the *test specification*: this contains all the necessary information for instantiating the test cases by unfolding the constraints.
6. Derive and evaluate the *test frames* by taking every allowable combination of categories, choices and constraints.
7. Generate the test scripts, i.e. the sequences of executable test cases.

The XML Schema provides an accurate representation of the input domain which leads quite naturally to the application of the Category Partition. In particular the subdivision of the input domain into functional units and the identification of categories can be done by exploiting the formalized representation of the XML Schema.

4 Automatic Instances Generation

In this section we briefly describe our original XML instances generation approach, which is called XML-based Partition Testing (XPT) [16]. A proof-of-concept tool called TAXI (Testing by Automatically generated XML Instances) implementing the proposed methodology is also described.

The XPT methodology is composed by two components: XML Schema Analysis (XSA), and Test Strategy Selection (TSS). The former, detailed in Section 4.1, implements a methodology for analyzing the constructs of the XML Schema and automatically generating the instances. The latter, described in Section 4.2, implements diverse test strategies useful both for selecting those parts of the XML Schema to be tested and for opportunely distributing the instances with respect to the Schema elements. These two phases work in agreement, as shown in Figure 1, to realize the application of the Category Partition method.

4.1 XML Schema Analyzer

In this section we introduce the functions realized by the XSA Component as schematized in Figure 1. Specifically XSA takes as an input the weighted version of the original XML Schema that is provided at the end of the first activity (details in Section 4.2) and foresees a *Preprocessor* activity in which the XML Schema constructs, like `all`, `simpleType`, `complexType` and so on, and the

[1] Note the usage of the same term "choice" both in XML schema syntax (written as <choice>) and in the CP method (written as *choice*), which is purely accidental.

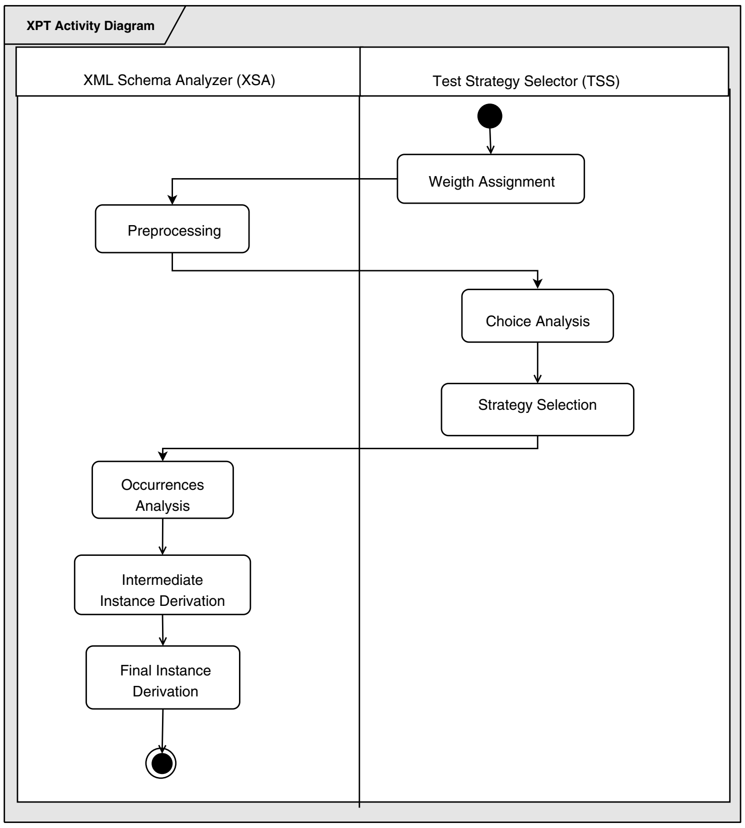
XPT Activity Diagram
XML Schema Analyzer (XSA)
Test Strategy Selector (TSS)
Weigth Assignment
Preprocessing
Choice Analysis
Strategy Selection
Occurrences Analysis
Intermediate Instance Derivation
Final Instance Derivation

Fig. 1. XPT main activities

shared elements, like `group`, `attributeGroup`, `ref`, and `type`, are analyzed and manipulated. The `choice` elements are excluded from the Preprocessor activity because they will be analyzed by the TSS component.

Considering, for instance, the `all` elements, one of the possible sequences of their children elements is randomly chosen [2] and used for generating instances; for each `group` element, instead, its body is copied wherever it is referenced. These preprocessing operations of course do not contribute to the definition of the test instances, but simplify their successive automatic derivation.

As detailed in Section 4.2 the next two activities by the TSS component have the purpose of: extracting the Functional Units (i.e. a list of subschemas) from the original XML Schema, by means of the analysis of `choice` elements, and selecting the test strategy that must be implemented (i.e. either covering a certain percentage of subschema functionalities or distributing a fixed number of instances among all the extracted subschemas, or a combination).

[2] A random selection algorithm which provides the elements ordering has been implemented for this purpose.

The implementation of the Category Partition methodology proceeds with the *Occurrences Analysis* activity, which analyzes the occurrences declared for each element in the subschema and, applying a boundary conditions strategy, derives the border values (*minOccurrences* and *maxOccurrences*) to be considered for the final instances generation.

The results of this activity are combined together during the last steps of the XPT methodology by deriving a set of intermediate instances structures. Values are given to the elements listed into each intermediate instance structure. For this purpose in the current version of TAXI a set of specific algorithms have been implemented to provide the required number of random values for each specific element type. In the current implementation, predefined values available in the Schema and various constraints (for instance `facets`), have been considered. Finally, according to the selected test strategy, the *Final instance derivation* activity produces the final set of instances, which corresponds to the test suite.

4.2 Test Strategy Selection

Testing is an essential, but expensive part of development. Hence test cases need to be prioritized, although generally it is not easy to decide on which parts of the application the testing effort should be concentrated and the amount of test cases to dedicate to each of them. The XML Schema representation of the input domain can help in this regard, by making it possible to implement a practical and automatic strategy for planning a suitable set of test instances.

The component of the XPT methodology which is in charge of this task is the Test Strategy Selection. It completes the implementation of the Category Partition by allowing for the selection of three specific test strategies. Referring to Figure 1 these include: *Weights Assignments*, which assigns weights to the children of the `choice` elements; *Choice Analysis*, which derives a set of subschemas the original XML Schema by means the analysis of the `choice` elements and first level elements; and *Strategy Selection*, which selects the test strategy to be implemented. We describe them in detail in the following.

Weights Assignments. The idea underneath the *Weights Assignments* activity is that the first level element or the children of the same `choice` may have not the same importance for instances derivation. There could be options rarely used or others having critical impact into the final instance derivation. Specifically considering with `choice` elements, according to their definition, only one child per time can appear into the set of final instances, hence from the user point of view the possibility of selecting those more important could be very attractive. He/she can pilot the automatic instance derivation forcing it to derive more instances including the most critical `choice` options.

The XML Schema does not provide the possibility of explicitly declaring the criticality of the diverse options, but often this information is implicitly left to the judgement and expertise of the human agent. The basic idea is that the XML Schema users are asked to make explicit this knowledge. In particular XPT explicitly requires to annotate each child of a `choice` element with a value, belonging to the [0,1] interval, representing its relative "importance" with respect to the other children of the same `choice`. This value, called the weight, must be assigned in such a manner that the sum of the weights associated to all the children of the same `choice` element is equal to 1. A node more critical has greater weight. Several criteria for assigning the importance factors could be adopted. Obviously this aspect in the proposed approach remains highly subjective, but here we are not going to provide a quick recipe on how numbers should be assigned. We only suggest expressing in quantitative terms the intuitions and information about the peculiarity and importance of the different options, considering that such weights will correspondingly affect the testing stage.

Once the weights have been assigned, XPT uses them to derive, for each option in the diverse `choice` elements, the relative importance factor, called the final weight, in terms of how risky is that child and how much effort should be put into the derivation of instances containing it. In a simplified version the final weight of every child is then computed as the product of the weights of all nodes on the complete path from the root to this node. Note that the sum of the final weights of the leaves is still equal to one.

Choice Analysis. As shown in Figure 1, after the *Preprocessor* activity the XPT methodology foresees the analysis of `choice` elements for deriving a set of subschema. These allow only one of the elements contained in their declaration to be present within a conforming instance. This means that for any alternative within a `choice` construct, a separate sub-XML Schema containing it can be derived. Stretching somehow the original meaning of a functional unit, each possible sub-schema is put in correspondence with the notion of a Category Partition functional unit. In other terms, in XPT functional units are meant as "domain units" and are thus assimilated to subsets of XML Schema elements that can originate correct testing instances by managing separate set of data inputs.

Obviously now the problem is the possible occurrence of several `choice`s within one schema, which gives rise to several possible combinations. In this case during the *Choice Analysis* activity as many subschemas as the number of the possible combinations of the children of the `choice` nodes are produced. In Figure 2 we report an example (for simplicity we omit from thew figure the assigned weights). In this case element a is a choice element, which includes a simple element b and another choice element c which has two children: x and y. In particular a transform to three sequence elements, one from element b, and two from the children element of c. In this way the original schema is divided into three subschemas.

During this operation the final weights previously derived are not modified: they will be used once derived the set of possible substructures. Using the final weights of the leaves in each substructure, it is possible to derive a unique value, called the *subtree weight*, useful for test strategy selection, as described in the next subsection. Specifically considering each substructure, starting from its root, the set of the partial subtree weights is normalized so that the sum of the subtree weights over the entire set of substructure is equal to 1.

Original XML Schema	1st Transformation	2nd Transformation	3rd Transformation
<element name="a">	<element name="a">	<element name="a">	<element name="a">
<complexType>	<complexType>	<complexType>	<complexType>
<choice>	<sequence>	<sequence>	<sequence>
<element	<element	<element	<element
name="b"…/>	name="b"…/>	name="c"…>	name="c"…>
<element	</sequence>	<complexType>	<complexType>
name="c"…>	</complexType>	<sequence>	<sequence>
<complexType>	</element>	<element	<element
<choice>		name="x"…/>	name="y"…/>
<element		</sequence>	</sequence>
name="x"…/>		</complexType>	</complexType>
<element		</element>	</element>
name="y"…/>		</sequence>	</sequence>
</choice>		</complexType>	</complexType>
</complexType>		</element>	</element>
</element>			
</choice>			
</complexType>			
</element>			

Fig. 2. Diverse subschema derived by the tag <`choice`>

Strategy Selection. Following the steps described so far each set of substructures has been defined, and a specific subtree weight has been assigned to each of them [3]. Now it is necessary to determine the test strategy to be adopted for test cases derivation. For this we consider three different situations: either a certain number of instances to be derived is fixed, or the percentage of functional coverage is chosen, or both are selected as a stopping rule. The first is the case in which a fixed number of instances must be derived from a specific XML Schema. In this case XPT derives the most suitable distribution of the derivable final instances among the subschemas previously defined. The second situation considers the occurrences with a certain percentage of subschemas, in other words the functionalities must covers a certain percentage of testing purposes. In this case XPT selects those subschemas that will be more suitable for testing purposes. Finally the last case is a mixed test strategy: it proposes a certain number of instances over a fixed percentage of functional coverage.

From a practical point of view, let us discuss the implications of each strategy:

– **Applying XPT with a fixed number of instances:** XPT strategy can be used to develop a fixed number NI of final instances out of the many that could be conceived starting from the original XML Schema. This could be in practice the case in which a finite set of test cases must be developed. Using the subtree weights associated to each substructure, the number of instances that will be automatically derived for each of them is calculated as NI times the subtree weight.
– **Applying XPT with a fixed functional coverage:** this corresponds to the case that a certain percentage of functional test coverage (e.g. 80%) is established as an exit criterion for testing. In this case considering the fixed coverage C, the selection of the substructures to be used can be derived by

[3] Of course if the original XML Schema did not include any `choice` element, at this point only one structure is available having *1* as subtree weights.

ordering in a decreasing manner the subtree weights, multiplying them times 100 and adding them together, starting from the heaviest ones, until a values greater than or equal to C is reached.

– **Applying XPT with a fixed functional coverage and number of instances:** in this case the above two strategies are combined. XPT first selects the proper substructures useful for reaching a certain percentage of functional coverage (as described above). Then considers the subtree weights of these selected subschemas and normalizes them so that their sum is still equal to 1. The new derived subtree weights are finally used for distributing among the selected substructures the fixed number of instances to be automatically derived.

5 The TAXI Tool

In this section we briefly describe the architecture of the TAXI tool, which implements the XPT strategy. The current version of TAXI can manage almost all elements of the XML Schema elements providing the set of required XML instances, even if some improvements are currently under implementation, such as the possibility of supporting namespaces or the usage of ontology for values assignment. TAXI will be released as open source code as soon as the development of the new added functionalities will terminate. Nevertheless in its current version it has been used as a proof-of-concept tool for verifying the efficiency and the applicability of the XPT methodology, providing encouraging results. TAXI takes an XML schema as input and parses it by using the W3C Document Object Model(DOM) [17]. It is mainly divided into five components (see Figure 3): User Interface, TSS, Preprocessor, SIP (Skeleton of Instances Producer), FIP (Final Instance Producer), and VP (Values Provider).

Specifically the User Interface manages the interaction with the user, who can influence and control the instance generation process accordingly with his/her specific requirements. By means of this components TAXI acquires the input to start the generation of the test case set. One of the tasks required to the user is therefore the selection of the XML Schema from which he/she wants to derive the valid instances and from this point ahead the generation proceeds automatically. User also needs to set the weights of the schema elements, and to select the test strategy. The weights as described in the previous section are used to represent the amount of test cases from different subtrees. Using weight and test strategy together TAXI can generate the proper amount of test cases from each subtree. The XML Schema is then passed to the Preprocessor component, which implements the preprocessor activities described in the previous subsection. The scope of this component is solving the tags `group`, `attributeGroup`, `ref`, `type`, `restriction`, `extension` and `all`. After this preprocessing stage, the input file is not a well-formed schema anymore, because the elements in the schema are not unique. In this so called "schema" there remain `sequence`,

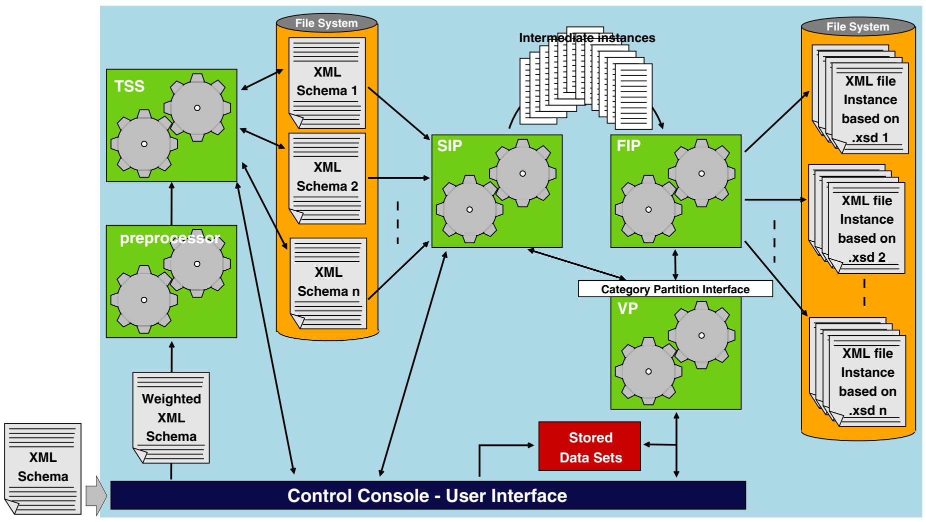
File System
XML Schema 1
XML Schema 2
XML Schema n
Intermediate instances
File System
XML file Instance based on .xsd 1
XML file Instance based on .xsd 2
XML file Instance based on .xsd n
TSS
SIP
FIP
preprocessor
Category Partition Interface
VP
Weighted XML Schema
XML Schema
Stored Data Sets
Control Console - User Interface

Fig. 3. Architecture of the tool TAXI

`choice` and simpleType elements. Then TAXI passes this "schema" to Test Strategy Selector. As seen, the first step of this component is `choice` solver, which produces multiple sub-schemas depending on the number of the `choice` constructs. At this point the component SIP (Skeleton of Instances Producer) retrieves and analyses each sub-schema, extracting from each element only the necessary information useful for the construction of the final instances. Meanwhile the weight of the child elements will be passed by the interface, and be attached to the sub-schema. Combining the weights with the test strategy, the total test cases can be calculated by TAXI automatically. In particular, when the condition $minOccurrences < maxOccurrences$ holds, collaborating with the component VP (Values Provider), it establishes the exact number of occurrences of each element. By using the collected data, the SIP component develops a set of skeleton files. These are mainly modified tree representations of the various sub-schemas in which special tags and instructions are introduced to make the final instances derivation easier. Specifically the number of skeletons to be produced results from the all possible combinations of the established occurrence values assigned to each element. Reflecting the activities described in the previous section the skeletons of instances so produced are finally analyzed by the FIP (Final Instance Producer) component. It uses the instructions provided by the SIP component in the skeleton, and collaborates with the VP component for receiving the correct values to be associated to each element. The final result is a set of instances, which are by construction conforming to the original schema and classified by sub-schemas. The VP (Values Provider) component has the task of providing the established occurrence of each element and the values to be assigned to each elements during the final instances derivation.

6 Considerations on Applicability of the Approach

The IMS Content Packaging Specification provides the functionality to describe and package learning materials, such as an individual course or a collection of courses, into interoperable, distributable packages. Content Packaging addresses the description, structure, and location of online learning materials and the definition of some particular content types.

As stated above the TAXI tool is still undergoing implementation, however in this section we provide a sample of its functionality, discussing its application within the e-Learning domain. In this domain, our purpose is to reduce the probability of having incorrect interactions among cooperating e-Learning tools: if the test cases are selected appropriately, the tools that pass all of them should be able to interoperate with the other tools that have been submitted to the same test campaign. Of course, in many cases the generation of all the possible instances could not be feasible given that the number could not be finite (consider for instance when an element has an unbounded *maxOccurences* attribute).

Learner Information Package is a IMS standard collection of information about a Learner (individual or group learners) or a Producer of learning content (creators, providers or vendors). As described in the IMS web site[4] the IMS Learner Information Package (IMS LIP) specification [18] addresses:

> "…the interoperability of internet-based Learner Information systems with other systems that support an Internet based learning environment. The intent of the specification is to define a set of packages that can be used to import data into and extract data from an IMS compliant Learner Information server, i.e. servers that in a eLearning environment collects data concerning pupils and/or eLearning content providers. A Learner Information server may exchange data with Learner Delivery systems or with other Learner Information servers. It is the responsibility of the Learner Information server to allow the owner of the learner information to define what part of the learner information can be shared with other systems. The core structures of the IMS LIP are based upon: accessibilities; activities; affiliations; competencies; goals; identifications; interests; qualifications, certifications and licences; relationship; security keys; and transcripts".

It is not difficult to understand the importance of conformance testing when such kind of open specifications are considered. The prefigured scenario is that different stakeholders will independently develop complex software systems that should all be able to take as input or generate in output conforming documents. The tacit assumption is that having considered an agreed specification they would be able to interoperate. Clearly this is far from being completely true. Even a simple XML based specification gives raise to infinite different XML instances. In particular it is possible to specify the same thing in many different ways but it is not difficult to find different parsers that will disagree on the

[4] http://www.imsglobal.org

conformance of an XML instance when the original specification imports many nested name spaces or complex tree structure.

In the next subsection we first analyse the dimensions of the feasible legal instances, for a given schema.

6.1 Number of Conforming Instances

Two different factors influence the variability of legal instances: instances can have different values for the same element (Data variability); or, instances can have different structures, i.e., they could contain different elements or different occurrences for the same element (Structural variability). For the case of structural variability three main reasons can be identified:

- the order of the elements in the instances (for instance the tag <`all`> leads to such kind of variability)
- the presence or otherwise of elements and/or attributes in the instances (for instance the tags <`choice`> or <`use`> lead to this situation)
- the number of possible occurrences of an element in the instances (due to the presence of attributes *minOccurrences* and *maxOccurences*)

Starting from these considerations and only focusing on structural variability, the number of correct instances foreseen by a certain XML Schema (represented as a tree structure), can be derived using the following formulas:

$$ChoiceNode = 2^{\sharp\{OptionalAttributes\}} \sum_{i=1}^{n} Subtree_i \tag{1}$$

$$AllNode = 2^{\sharp\{OptionalAttributes\}} n! \prod_{i=1}^{n} Subtree_i \tag{2}$$

$$SequenceNode = \prod_{i=1}^{n} Subtree_i \tag{3}$$

$$minMaxOccurNode = 2^{\sharp\{OptionalAttributes\}} \sum_{i=minOccur}^{maxOccur} (\prod_{j=1}^{n} Subtree_j)^i \tag{4}$$

$$LeafNode = 2^{\sharp\{OptionalAttributes\}} (maxOccur - minOccur) \tag{5}$$

In the formulas above, the variable n indicates the number of different subtrees of a given node. The name of the left member of a formula indicates when to apply it. For instance if the node contains a <`choice`> tag the formula to apply will be the first. In order to calculate the number of possible instances a simple visit of the XML Schema tree is sufficient. However in the general case this number cannot be calculated when there are unbounded occurrences of an element or loops in the structure of a subtree: for example, a `complexType` that in one of the corresponding subtrees contains an element of the same type.

Just to give a flavor we calculated the number of possible structurally different instances that can be generated starting from the LIP XML Schema [18]. According to the formula, under the restrictive assumption that no *maxOccurrences* attribute can assume values greater than three, from the schema [18] we calculate that there are 78912 valid instances that can be generated from the main element "product". To reduce the number of equivalent instances, we use boundary conditions strategy: when the minOccurences < maxOccurences, we use only the minimum value and maximum value to do the combination. With this simplified method, 35200 valid instances are obtained from the given schema.

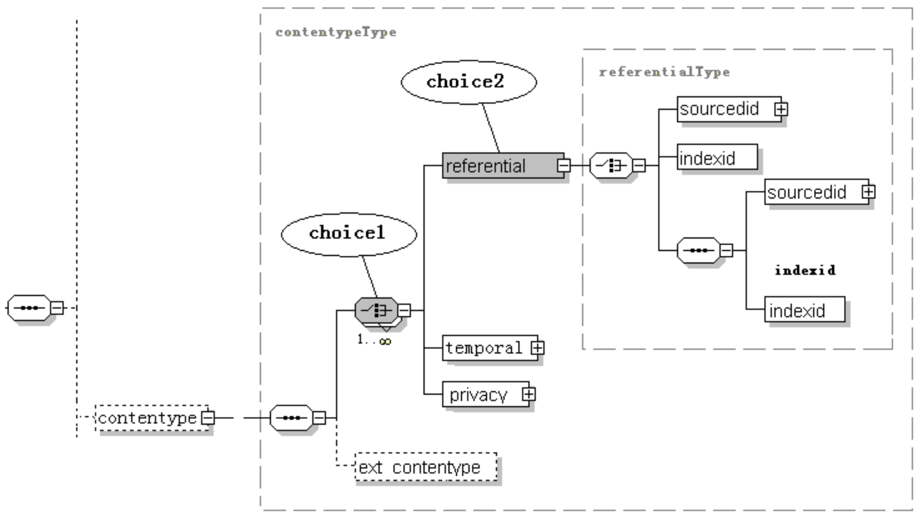

Fig. 4. Partial schema tree

This simple result can probably provide the most intuitive reason to suggest the use of a systematic approach to the generation of XML instances for testing purpose. Given that only a small part of the instances can be used for testing purpose it is absolutely necessary to apply a systematic strategy for the derivation of the test cases. The strategy should permit to focus on conditions that the tester could judge particularly critical in a specific setting. For instance for a particular application the tester could judge the variability on the number of occurrences more important than the order of the elements. Considering the schema that was presented in Fig 4, there are two `choice` elements: "choice1" is the child element of `contentype`, "choice2" is element "referential", which is a child element of "choice1". We set the weight for "choice1" first. There are three child elements in this complexType, we set the weight of "referential" as 0.5, the weight of "temporal" and "privacy" are 0.3 and 0.2. Then consider the weight for another choice element "referential" which has three child elements as well. We set the weight of "sourceid" as 0.3, "indexid" as 0.2, and the other one as 0.5. After solving `choice` 5 subtrees are derived. TAXI can calculate the weights for each subtree automatically according to the weights of choice nodes. The weights of these five sub-schemas are given below.

– The weight of subtree that includes "soureid" is 0.15
– The weight of subtree that includes "indexid" is 0.10
– The weight of subtree that includes "soureid" and "indexid" is 0.25
– The weight of subtree that includes "temporal" is 0.3
– The weight of subtree that includes "privacy" is 0.2

6.2 XPT vs. Random Generation

The possibility of automatically deriving instances from a XML Schema is an emerging problem in many fields of application. As mentioned in Section 2 some tools have been implemented to this purpose. However all of them rely on the random generation of instances, and do not implement any systematic and specific testing strategies. In this section we want to compare the performance of such a kind of existing tools with our tool TAXI. Specifically we select XMLSpy [11], which is an industrial standard XML development environment for modeling, editing, debugging, and transforming all XML technologies. For generating the instances, XMLSpy asks the user to perform some preliminary configuration settings, including: filling elements and attributes with data, whether generating the non-mandatory elements and attributes, generating a priori selection of mandatory `choice` element or not, and how many elements should be generated when *maxOccurrences* is more than one. Thus XMLSpy is different from TAXI both in the strategy implemented and in the typology of instances obtained. We list the mains aspects that characterize the two tools in the following.

1. The amount of instances: XMLSpy generates several configurations, but from each of them only one instance can be derived. TAXI has the capability of deriving large quantity of instances covering systematically all the aspects of a specific XML schema.
2. The value of elements: XMLSpy always gives a same fixed value for each data type. For instance the `<date>` type is fixed to "1967-08-13", and `<string>` type to "string". TAXI has the possibility of declaring a specific set of values for each data type or randomly generating as many values as required.
3. The solution of `<all>` elements: XMLSpy does not make difference in deriving instances when there is a `<all>` or `<sequence>` element, i.e. in the two cases the derived instances will have the same structure. TAXI generates all the possible combinations of the `<all>` children element, and then randomly selects one from them.
4. The solution of `<choice>` elements: In presence of a specific request from the user, XMLSpy can get instances with the first child element of `<choice>` element, otherwise XMLSpy leaves the content of choice element as empty. TAXI derives diverse instances for each of the `<choice>`'s children elements, covering in this manner all the possibilities.

5. The solution of occurrences: in XMLSpy all the values of occurrences must be fixed between 1 to 99. TAXI leaves the user both the possibility of declaring the values of occurrences or using the boundary values. In case of unbounded occurrences, if the user does not set a preference value, TAXI adopts a prefixed bound. The occurrences values are then combined to get instances with variation structures.

Considering a complex schema, TAXI nearly generates all possible combinations of complex elements and occurrences, and each instance has different values inside, while the instances from XMLSpy vary only in the amount of repeated elements. Concluding despite the good performance of XMLSpy, for the instance generation this tool applies a quite simple algorithm, which gives only few flexibility to the user and does not attempt to cover all the input domain. From the tester's point of view the derived instance cannot cover all the declared schema elements and consequently the functionalities of the application to be tested. Thus it could be claimed that TAXI is able to provide a test strategy, which is more comprehensive and covers all weaknesses of XMLSpy.

7 Conclusions

We have introduced the XPT approach for the systematic derivation of XML Instances from a XML Schema. XPT applies to the XML notation a well-known method for software black-box testing. Given the pervasiveness of XML in web-based and distributed applications, we are convinced that the proposed method can be very useful to check the quality of applications via a rigorous test campaign. In generak, we are interested in generating both valid and invalid instances (the latter for robustness test). On the tester's side, XPT targets the long-standing dream of automating the generation of test cases for black-box testing, which is routinely done by expert testers that analyse specifications of the input domain written in natural or semiformal language. If the input is formalized into XML Schema, then XPT can provide a much more systematic and cheaper strategy. The work we have described is still undergoing implementation. We will continue investigating the applicability to real-world case studies, in particular within the e-Learning domain. The most challenging issue that comes out from the investigation in this paper is the infeasibly high number of test instances that would be generated, therefore the identification and implementation of sensible heuristic to reduce the generated instances is compelling.

References

1. W3CXML: W3cxml. http://www.w3.org/XML/ (1996)
2. W3CXMLSchema: W3c xmlschema. http://www.w3.org/XML/Schema (1998)
3. XMLTestSuite: Extensible markup language (xml) conformance test suites. http://www.w3.org/XML/Test/ (2005)
4. NIST: Software diagnostics&conformance testing division: Web technologies. http://xw2k.sdct.itl.nist.gov/brady/xml/index.asp (2003)

5. RTTS: Rtts: Proven xml testing strategy. http://www.rttsweb.com/services/index.cfm (nd)
6. SQC: Xml schema quality checker. http://www.alphaworks.ibm.com/tech/xmlsqc (2001)
7. W3CXMLValidator: W3c validator for xml schema. http://www.w3.org/2001/03/webdata/xsv (2001)
8. XMLJudge: Xml judge. http://www.topologi.com/products/utilities/xmljudge.html (nd)
9. EasyCheXML: Easychexml. http://www.stonebroom.com/xmlcheck.htm (nd)
10. Li, J.B., Miller, J. In: Testing the Semantics of W3C XML Schema. COMPSAC 2005 (2005) 443 – 448
11. XMLSpy: Xml spy. http://www.altova.com/products_ide.html (2005)
12. Toxgene: Toxgene. http://www.cs.toronto.edu/tox/toxgene/ (2005)
13. SunXMLInstanceGenerator: Sun xml instance generator. http://wwws.sun.com/software/xml/developers/instancegenerator/index.html (2003)
14. Ostrand, T., Balcer, M.: The category-partition method for specifying and generating functional tests. Communications of ACM **31**(6) (1988)
15. Basanieri, F., Bertolino, A., Marchetti, E.: The cow_suite approach to planning and deriving test suites in uml projects. In: Proc. Fifth International Conference on the Unified Modeling Language UML 2002, LNCS 2460, Dresden, Germany (2002) 383–397
16. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Partition testing from xml schema. Technical report ISTI-45/2005 (2005)
17. DocumentObjectModel: Document object model. http://www.w3.org/DOM/ (2005)
18. AAVV: IMS learning information package v.1.0.1. On-line at: http://www.imsglobal.org/content/packaging/cpv1p2pd/imscp_oviewv1p2pd.html (2005)