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 propose the XPT approach for the automated generation of XML Instances which conform (or do not conform) to a given XML Schema. XPT can be a very useful instrument for testing those applications that expect in input the XML instances, and have been fruitfully exploited in the e-Learning domain. Indeed, we generate a set of instances by systematically considering the possible combinations of elements within the schema, according to the well known Category-Partition strategy for black box testing. Here we trial the TAXI proof-of-concept tool, that implements the XPT approach, to the IMS Content Packaging XML Schema, and briefly discuss the results.

1 Introduction

More and more complex software systems are developed according to a modular architecture, within which precise features can be identified and separately implemented. Main objective of "componetization" is to permit the development of the different features by diverse stakeholders still having the possibility of integrating the subsystem into a unique working system. Nevertheless the integration phase clearly require to definition of clearly and checkable format the exchanged data.

Certainly one of the most important innovation 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 according to the rules of the referred XML Schema, represent the allowed naming and structure of data for component interaction and for service requests.

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

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 interpretation by various developer. 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 wants to introduce the reader to the systematic generation of XML instances and on corresponding advantages. At the same time the paper reports a first evaluation of the approach to the generation of instances for the IMS Learning Information Package specification. 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 generate a set of XML instances for the black box testing of a component, whose expected input conforms to the taken schema.

In the remainder of this paper we discuss some related works in Section 2, then Section 3 provides a short introduction on the proposed strategy and on the tool implementing it. Section 4 wants to give quantitative motivations to the application of a systematic approach, then in Section 4.2 a simple comparison with another possible approach is presented. Some conclusions are finally discussed in Section 5.

2 Related work

Notwithstanding the intense production of XML-based methods and tools in the latest years, we are not aware of XML-based test approaches comparable to ours. Our interest is in automatically generating a comprehensive suite of XML instances from a given XML Schema, for the purpose of testing applications that

will use such XML instances as input. Instead, what existing "test tools" based on XML do, 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 called *well-formedness*, aimed at verifying that the file structure and its elements possess specified characteristics, without which the tested file cannot be even classified as an XML file. For this aim, 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, several validators exist for checking the syntax and the structure of the W3C XML Schema document, and the definition of the systems (for instance, [6], [7]. [8] and [9], [10])

The third group enclose tools for automated instance generation based on XML schema. Relevant elements of the group are [11], [12] and [13]. Nevertheless all the available tools for XML instances generation only implement random or ad hoc generation. 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.

Finally, regarding Category Partition, which has been previously applied also by some of the authors of this paper [14], so far no proposal has really succeeded in pushing the widespread adoption of automated black box testing as it would deserve. We think that the widespread acceptance of XML, and its pragmatic flavor, associated to the Category Partition methodology could finally be the winning instruments to do so.

3 Automatic Instances Generation

In this section we briefly describe an original XML instances generation approach, called XML-based Partition Testing (XPT) [15]. At the same time a proof of concept tool called TAXI (Testing by Automatically generated XML Instances) implementing the proposed methodology is described.

The XPT methodlogy is composed by two different phases: XML Schema Analysis(XSA) and Test Strategy Selection (TSS). The former, detailed in Section 3.1, implements a methodology for analyzing the constructs of the XML Schema automatically generating instances. The latter, described in Section 3.2, implements diverse test strategies useful for both selecting the parts of the XML Schema to be tested and opportunely distributing the instances on the Schema.

These two phases work in agreement, as shown in Figure 1, to realize the application of a partition testing technique, as defined in [16], which constitutes the overall basis on which the whole XPT approach relies on.

Introduced in the late 80's and today widely known and used, the Category Partition (CP) [16] 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: processing. They are not yet a list of test cases, but contain all the necessary information for instantiating them 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 leads quite naturally to the application of the Category Partition, since it provides an accurate representation of the input domain. 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.

3.1 XML Schema Analyzer

In this section we introduce the functionalities in charge of the XSA Component as schematize in Figure 1. The XML Schema Analyzer collaborates with the Test Strategy Selector part as visualized in Figure 1. Specifically XSA takes the weighted version of the original XML Schema provided at the end of the first activity as an input (details in Section 3.2) and foresees a *Preprocessor* activity in which the XML Schema constructs, like `all`,`simpleType`,`ComplexType` and so on, and the shared elements, like `group`, `attributeGroup`, `ref type` are analyzed and manipulated. The `choice` elements are the only ones excluded from the preprocessor activity because analyzed by the TSS component.

Considering, for instance, the `all` elements one of the possible sequence of their children elements is randomly chosen, [2] and used for generating instances,

[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.

[2] A random selection algorithm which provide the elements order has been implemented for this purpose

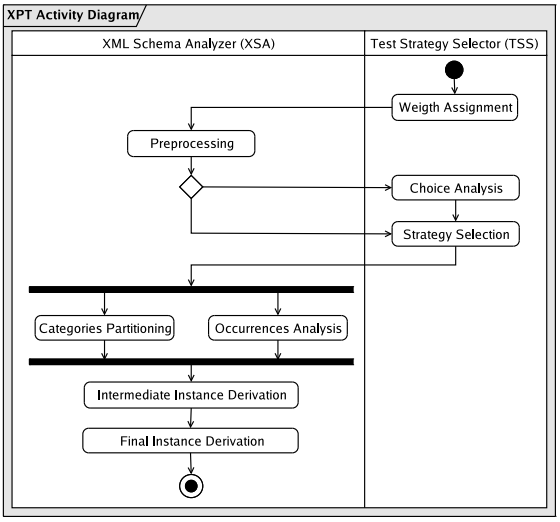

Fig. 1. XPT main activities

while for `group` its body is copied in each element which refers them. These preprocessing operations of course do not contribute to the definition of the test instances, but simplifies their successive automatic derivation.

As will be detailed in Section 3.2 the next two activities have the purpose of: extracting the Functional Units (i.e. a list of subschemas) from the original XML Schema, by mean 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 between all the extracted subschema, or both) .

The implementation of the Category Partition methodology proceeds with the activity called *Categories Partitioning*, and the *Occurrences Analysis* which partition the categories into *choices* and determine constraints among them, respectively. In particular the former takes as an input the set of set of subschema selected by the test strategy and for each of them it extracts from the involved Schema elements the required information for generating the intermediate structures that will be used for the final instances description. The latter activity analyzes the occurrences declared for each element in the subschema and, applying a boundary condition strategy, derives the border values (minOccurrences and maxOccurrences) to be considered for the final instances generation.

The results of this two activities are combined together during the last but two and last but one steps of the Category Partition methodology by deriving a set of intermediate instances structure, each one derived by the combinations of the elements couple occurrences. Finally, accordingly with the test strategy selected and by giving values to the element listed into each intermediate instance structure, the *Final instance derivation* activity produces the final set of instances, which correspond to the test suite. 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 their implementation, predefined values available in the Schema and various constraints (for instance facets), have been also considered.

3.2 Testing Strategy Selection

As said before the XML Schema represents in a clear way the overall structure of the input domain. A part from the advantages mentioned in the previous subsection, this is an enormous benefit for test planning.The testing phase is an expensive but essential part of development, which must be well-organized and defined. Generally, it is not easy to decide on which parts the testing effort should be concentrated and the amount of test cases to dedicate to each of them. Wrong decisions could increase the overall cost an the completion time of the testing phase Thanks to the XML Schema representation is possible to implement an integrated, practical and automatic strategy planning a suitable set of instances, i.e. test cases.

The part of the XPT methodology which is in charge of this task is the Testing Strategy Selection. It completes the implementation of the Category Partition and lets the selection of three specific test strategies to be applied. Referring to Figure 1 it includes three activities *Weights Assignments*, *Choice Analysis*, *Strategy selection* which respectively: assign weights to the children of the `choice` elements; derive a set of substructure from the original XML Schema by means the analysis of the textttchoice elements; select which test strategy must be implemented; We describe them in detail in the following subsection. Of course if the Schema does not include any `choice` element the first two activities are not executed.

Weights Assignments The idea underneath the *Weights Assignments* activity is that 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. Because according with the definition of `choice` element only one child per time can appear into the set of final instances, from the user point of view the possibility of selecting those more important could be very attractive. He/She can pilots the automatic instance derivation forcing it to derive more instances including the most critical `choice` options.

The XML Schema doesn't provide the possibility of explicitly declare the criticality of the diverse options, but often this information is implicitly left to the sensibility and expertise of the people deriving instances. The basic idea here is asking the XML Schema expert or user to make explicit this knowledge. For this we provide them with a systematic strategy in order to use such information for test planning. In particular XPT explicitly requests to annotate each children 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. The more critical a node the greater its 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 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.

The problem is of course 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 number of the possible combination of the children of the `choice` nodes are produced. In Figure 2, we report an example in which for aim of simplicity we omit 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. For aim of simplicity we avoid to represent the weight assignments for each node. In particular a transform to three sequence elements, one from b, and two from the children selection of c. By 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 once derived the set of possible substructures. Using the final weights of the leaves in each substructure, it is possible to derive a unique values, called *subtree weights*, useful for test strategy selection as will be described in the next subsection. Specifically considering each substructure, starting form its root the set of the deepest node having a final weights is derived. These numbers are then summed together for obtaining a partial subtree weight. The final subtree weight is then derived dividing this value for the overall sum of the partial subtree weight of each substructure. This operation normalize the

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

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

set of the partial subtree weights so that the sum of the subtree weights over the entire set of substructure is equal to 1.

Test Strategy Selection Following the steps described so far each a set of substructure has been defined and a specific subtree weights assigned to each of them [3]. Now it is necessary to determine test strategy to adopt for test case derivation. For this we consider three different situations: either a certain number of instance to be derived is fixed, or the percentage of functional coverage is chosen, or both is 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 allows to derive the most suitable distribution of the derivable final instances cases among the subschemas previously defined. The second situation considered occurs when a certain percentage of subschema, or in other words functionalities, must be covered for testing purposes. In this case XPT opportunely select those subschema that will be more suitable for testin purposes. Finally in the last case a mixed test strategy is proposed where a certain number of instances over a fixed percentage of functional coverage is considered.

As a practical point of view that mentioned above is applied as:

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

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

testing. In this case considering the fixed coverage C, the selection of the substructure to be used can be derived by 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 or equal to C is reached.
– **Applying XPT with a fixed functional coverage and number of instances** In this case the above mentioned 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 substructure the fixed number of instance to be automatically derived.

3.3 Testing by Automatically generated XML Instances

In this section we briefly describe the architecture of the TAXI tool, which has the purpose of implementing 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 concepts tool for verifying the efficiency and the applicability of the XPT methodology, providing encouraging results. TAXI, which takes an XML schema as an input and parses it by using the W3C Document Object Model(DOM) [17], is mainly divided into five components, (Figure 3): User Interface, test strategy selector, Preprocessor, SIP (Skeleton of Instances Producer), FIP (Final Instance Producer), 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 task that required to the user is therefore the selection of the XML Schema from which he/she wants to derive the valid instance and from this point ahead the generation proceeds automatically. After XML schema input user need to set the test weight of the schema, and select the test strategy. The weight as described in the previous section is used to represent the amount of test cases from different subtree. 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 that described in the previous subsection. The scope of this component is solving tags `group`, `attributeGroup`, `ref`, `type`, `restriction`, `extension` and `all`. After preprocessor the input file is not a well-formed schema, because the elements in the schema are not unique. In this so called "schema" only `sequence`, `choice` and simpleType elements are remained. Then TAXI pass this "schema" to test strategy selector. The first step

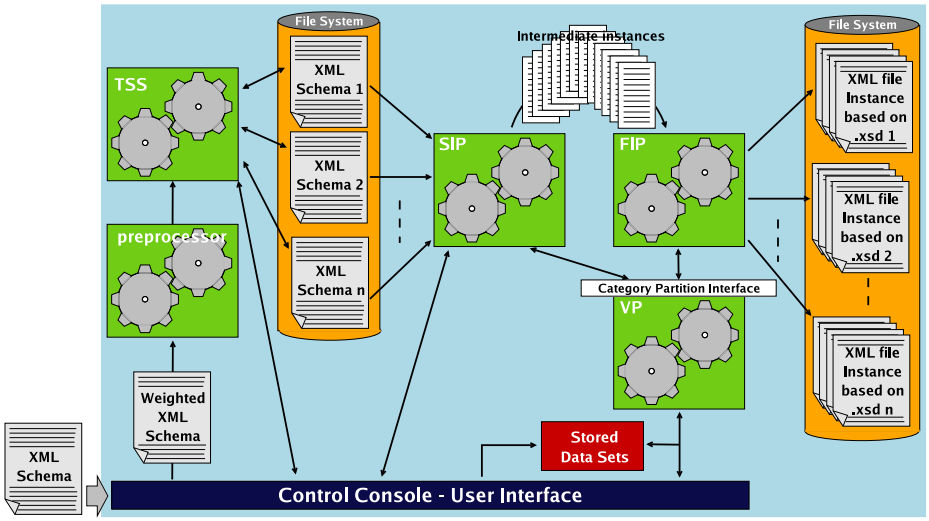

Fig. 3. Architecture of the tool TAXI

of this component is choice solver, which is producing multiple sub-schemas depending on the number of the `choice` constructs. At this point the component SIP (Skeleton of Instances Producer) retrieves and analysis 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 passed from the interface, and be attached to the sub-schema. Combining the weight 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 subschema in which special tags and instructions are introduced to make the final instance 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 instance derivation.

4 Considerations on Real Application of the Approach

As stated above TAXI is still under implementation, however in this section we want to give a flavour of its functionality showing the generation of correct content packaging XML instances [4]. The purpose is to reduce the probability of having incorrect interactions among cooperating e-Learning tools: if the test cases are selected appropriately, a tool that pass all of them should be able to interoperate with the other tools that have been submitted to the same test campaign. As can easily imagined, 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 elements have an unbounded maxOccurence attribute). Two different factors influences the variability of correct instances: instances can have different values for the same elements (Data variability);instances can have a different structure, i.e. they could contains different element or different occurrences for the same elements (Structural variability). In case of structural variability we can have three main reasons that lead to have structurally different but still correct instances:

- the order of the elements in the instances (for instance the tag <`all`> lead to such kind of variability)
- the presence or not 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 this considerations and only focusing on structural differences, the number of correct instances foreseen by a certain XML Schema (represented as a tree strucuture), can be derived using the following formula:

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

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

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

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

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

[4] It is worth to note that the developed tool can only generate part of a real Content Packaging instance as the XML files. Nevertheless it can be integrated in a more complex tool that taking in input a particular XML instance construct a correct CP file

In the formula the variable n indicate the number of different subtrees of a given node. The name of the left member of the formula indicate 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. Such as complex type 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]. As the description of the formula, under the restrictive assumptions that no maxOccurrences attribute can assume values greater than three since in the case study if the maxOccurs is "unbounded" or greater than three then it will be modified to default value "3", from the schema [18] there are 78912 valid instances that can be generated from the main element "product". In the case study in order to reduce the equal instances, we use boundary condition strategy to select the occurrences, so that if the minOccurs "¡" maxOccurs, only the minimum value and maximum value will be taken and be used to do the combination. By using this simplified method we obtained 35200 valid instances from the given schema.

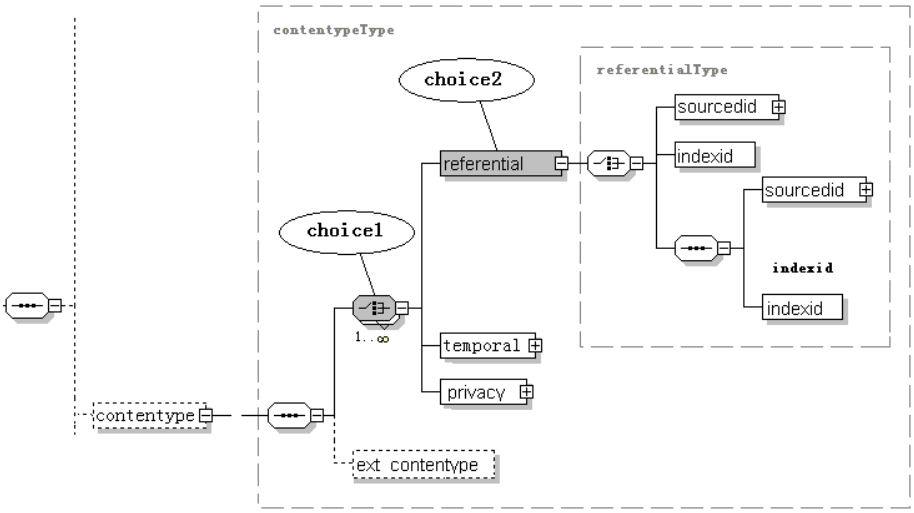

Fig. 4. Partial schema tree

This 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 possible instances can be used for testing purpose, test strategy selector 3.2 is absolutely necessary. Using test strategy, we need set the weights for choice elements after input the schema. From fig 4 which is a part of the schema tree, it is clear to see that there are two choice

elements in this schema. "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 we set 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 choice solver 5 subtrees are derived. TAXI can calculate the weights for each subtree automatically according the weight of each child element of choice node. The weights for 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

Concerning the effectiveness of the approach in discovering bugs, unfortunately we did not have the possibility yet to provide the generated instances to the software accepting it as input. Nevertheless the proposed approach is in some way a useful way of applying the Category Partition methodology so we expect that the approach will show similar properties with respect to Categoly Partition.

4.1 Application of the XML instance derivation strategy to the IMS Learning Information Package

Learner Information is a 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[5] 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".

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

It is not difficult to understand the importance of conformance testing when such kind of open specifications are considered. The prefigured scenario foresee that different stakeholder will start to indipendently develop complex software system that will be able to take as input or generate in output conformant document. The tacit assumption is that having considered an agreed specification they would be able to interoperate. Clearly this is far from being completely true. The problem is that even a simple XML based specification give raise to an infinite possible different XML instances and it is not difficult to imagine that a bug can be introduced in the implementation of a system, affectin then its ability to interoperate. Moreover a XML Schema specification suffers per se of problems due to its inherent complexity. In particular it is possible to specify the same thing in many different way but it is not difficult to find diffenrent parser that will disagree on the conformance of an XML instance when the starting specification import many nested name spaces or complex tree structure.

Just to give a flavour we calculated the number of possible structurally different instances that can be generated starting from the IMS LIP XML Schema [18]. We obtained the impressive number of 5,352,761,232,000 possible correct instances under the restrictive assumptions that no maxOccurrences attribute can assume values greater than three and unfolding all the loops until the first level. 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 possible 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.

4.2 XPT vs Random Generation

The possibility of automatically deriving instances from a XML Schema an emerging problem in many fieds of application and some tool have been implemented to this purpose as mentioned in 2. However all of them rely on the random instance gneration, and do not implement any systemstic and specific tesing strategies. In this section we want to compare the performance of such a kind of toolwith 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 max-Occurs is more than one. Thus XMLSpy is different from TAXI both in the strategy implemented and in thetypology of instances obtained. We list in the following the mains aspects that characterize this two applications.

1. The amount of instances: XMLSpy generates several configurations, but and from each of them only one instance can be derived. TAXI has the capability of deriving large quantity of instances covering sistematically of the aspect of a specific XMLschema.
2. The value of elements: XMLSpy always gives a 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 generate as many values as required.
3. The solution of `<all>` elements: XMLSpy does not make difference in deriving instances wherever there is a `<all>` or `<sequence>` element, i.e. in both the cases the derived instance will have the same structure. TAXI generates all the possible combination of the `<all>`'s child element, and then randoly select one from them.
4. The solution of `<choice>` elements: In presence of a specific request from the user the XMLSpy can get instances nly 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 child elements, covering in this manner all the possibilities.
5. The solution of occurrences: in XMLSpy the all values of occurrence must be fixed between 1 to 99. TAXI leave the user both the possibility of declearing the values of occurrences or using the some boundary values. In case of unbounded occurrences TAXI, if there is not a user value, the tool adopts a prefixed bound. The occurrences values are then combined for get instances with variation in structures

Adopting the Content Packaging Schema with both the tool we obtain a maximum number of 402,472,960 instances from TAXI, and only 102 instances from XMLSpy. The instances from TAXI cover with nearly possible combinations of complex elements and occurrences, and each instance has the 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 more comprehensive testing strategy, which covers all weaknesses of XMLSpy.

5 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 fact, 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 investigate 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. Basanieri, F., Bertolino, A., Marchetti, E.: The cow_suite approach to planning and deriving test suites in uml projects. In 2460, L., ed.: In Proceedings Fifth International Conference on the Unified Modeling Language UML 2002, Dresden, Germany (2002) 383–397
15. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Partition testing from xml schema. submitted to IEEE Software (2006)
16. Ostrand, T., Balcer, M.: The category-partition method for specifying and generating functional tests. Communications of ACM **31**(6) (1988)
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)