Partition Testing from XML Schema

Antonia Bertolino, Jinghua Gao, Eda Marchetti, Andrea Polini
ISTI-CNR
Via Moruzzi, 1
Pisa (Italy)
{antonia.bertolino, jinghua.gao, eda.marchetti, andrea.polini}@isti.cnr.it

Testing is a crucial part of the software development process, which strongly affects the quality and reliability of the delivered product. As it consumes a huge part of the effort required in software production, approaches that can effectively systematize and automate software testing are ceaselessly sought after both in industry and academy.

It was already in the earliest days of the software testing discipline that the intuitive term of "partition testing" has been coined (see, e.g., Richardson and Clarke[1]) to refer to a broad class of test criteria that systematically sample a program's input domain. The basic idea behind partition testing is that the input domain can be divided into subdomains, such that, for testing purposes, within each of them the program can be assumed to "behave the same", i.e., for every point within a subdomain the program either succeeds or fails. This assumption is what allows for getting a finite set of tests out of the infinite execution domain, since, based on it, from observing its execution on one or few points within each subdomain, the program's behaviour over the whole domain can be (hypothetically) deduced.

The principle of input partitioning is fairly general, and can be indifferently applied at any testing level, also including the unit testing stage. Unit testing is widely acknowledged as a key step in ensuring the final system quality. At this level, the single units composing a wider system are tested before integration, for verifying that they satisfy their functional specification. In modern days, the emergence of compositional paradigms to complex systems production, such as the Component-Based Development[2], has boosted the importance of unit testing, here meant as the thorough testing of the reusable units of composition.

In the years, a wealth of partition test techniques has been introduced, both black-box and white-box. In this paper, we focus on black-box partition testing techniques, variously implementing the common underlying strategy of: *i)* looking at a program input variables, *ii)* identifying relevant classes of input values, and *iii)* choosing some representative test input values for each identified class. In such a scheme, the degree of automation which is achievable for test generation clearly depends on the formalization level at which the input domain is described: although several researchers have early attempted to exploit formal specifications, the prevalent approach to input domain partitioning remains a semiformal one, and is well exemplified by the Category Partition (CP) method. CP provides a stepwise intuitive methodology for the testing of "functional units" from their specifications written in structured, semiformal language, and has inspired the development of a large number of (semi)automated unit test methodologies.

Nowadays, the eXtensible Markup Language (XML)[3] has established itself as the de facto standard format for specifying and exchanging data and documents in almost any computer and web application. Paired with XML diffusion, the XML Schema[4] has spread up as the notation for formally describing what constitutes an agreed valid XML document within an application domain. One or more XML Schemas are used for expressing basic structural rules and complex restrictions of the diverse data and parameters that units/components exchange with each other, while XML documents, also called 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.

Describing input data via XML Schema yields important positive effects for improving interoperability between system components. Establishing in fact a formalized agreement on the format of data exchange supports the application of testing strategies for checking the local data structures and the interfaces used by the different components. The recent literature collects several contributions dealing with testing *of* XML Schemas, as opposed to our approach of testing *from* XML Schema: we suppose that a (valid) XML Schema has been defined, and we exploit that schema for automating the testing of the components that refer to it.

To the best of our knowledge this is the first work in this direction: we propose to apply the well established input domain partition testing technology onto XML Schema, the nowadays universal language for data representation, so that, starting from the latter, functional test cases, expressed in the form of XML instances, can be systematically and automatically derived. Our proposed method is called XML-based Partition Testing, or XPT.

Indeed, the adoption of the XML Schema lends itself quite naturally to the application of partition testing, as we will show in the following. In fact, the XML Schema provides an accurate representation of the input domain, or in other words of the data and parameters that components will exchange, into a format suitable for automated processing, which is clearly a big advantage for testing. Thus the subdivision of the input domain into subdomains, following the basic principle of partition testing, can be done exploiting the formalized representation of the XML Schema: from the diverse subdomains identified, the application of partition testing then amounts to the systematic derivation of a set of XML instances.

A nice peculiarity of applying partition testing to XML Schema is that by construction the derived test instances correspond to *true partitions*, i.e., the input subdomains do not overlap, because each of them focuses only on specific data entries. This is a desirable characteristic for partition testing, but hardly obtained when the subdomains are derived from functional specifications.

Some tools are being proposed for deriving XML instances from a XML Schema, like XML-XIG[5]. Such tools solve only one piece of the wider problem we address: a method for systematically applying partition testing on a XML Schema and deriving meaningful test cases, in the form of XML instances.

Clearly, a key to success for the proposed XPT approach, as for any testing strategy, is the capability of automation: a proof of concept tool, called TAXI (Testing by Automatically generated XML Instances), is described for illustrating how XPT can be automated. TAXI takes in input an XML Schema and automatically generates a set of XML Instances for the black box testing of a component whose input conforms to the taken schema.

Interoperating via XML

The XML[3] is a cross-platform, text-based and extensible language, which provides a standard format for storing and exchanging documents between computer applications (and not only). XML is the W3C endorsed standard for document mark-up; in simple terms, it defines a generic syntax used to mark up data with intuitive, human-readable tags.

XML can store and organize any kinds of information in a textual form. The information is represented as a hierarchy of elements with names, optional attributes and contents, which can be defined by the developers. With Unicode as its standard character set, the XML supports many

kinds of writing systems and symbols, and it can be edited with any kind of editor, from a standard text editor to a visual development environment. Besides, XML is easily combined with style sheets to create formatted documents in any desired style. Because of so many benefits, XML is fast becoming the widest used format for data interchange between a system's components, and also on the Web.

A well-formed XML document obeys XML syntax rules. A grammar that defines the rules and the structure of a XML document is called a XML Schema[4]. A XML Schema is a schema that is itself written in XML. It defines the rules of the elements, attributes and other items of XML documents. XML Schema can support data types and namespaces (a namespace is a collection of names, identified by URI references), and can be easily extended. XML Schema was released by the W3C and became an official W3C recommendation in May 2001. Now it is supported by a wide range of parsers and XML applications.

In the table below, we show the definitions of the main XML Schema elements[4].

Table 1 : Subset of XML Schema elements

Element	Explanation
simpleType	Defines a simple type that can contain information and constraints about the allowed values or text-only elements. It cannot contain other nested elements.
complexType	Defines a complex type element that may contain other elements and have attributes.
attribute	Defines an attribute that can be used for the complex type elements.
choice	Specifies that only one of the child elements can be present in a document.
sequence	Specifies that instances of the child elements must appear in a same order as in the declaration. Each child element can be absent or occur any number of times.
all	Specifies that the child elements may appear in any order. Each element can be absent or occur at most once.
group	Creates a reusable component of elements that can be used in complex type definitions.
attributeGroup	Defines a group of attribute references or declarations to be used in complex type definitions.
simpleContent	Defines an element that contains the extensions or restrictions on a simple type or text-only complex type.
complexContent	Specifies a complex type element from extending or restricting another complex type.

XML documents typically contain an element that includes text, attributes, and other elements that, in turn, may include elements, text and attributes. This structure can be described as a tree structure. All items in the XML documents hold positions in the overall tree.

XML Partition Testing

We now introduce the XPT approach for the application of the Category Partition test method (see sidebar description) to a XML Schema description of the input domain.
The first step in CP is the identification of the functional units: for this, we consider the construct `<choice>`[i], which allows only one of the elements contained in its declaration to be present within a

[i] 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.

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 CP 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. For instance Table 2 reports the basic transformation rule adopted for splitting the original Schema into its functional units in presence of multiple <choice>s: note that after the transformation each functional unit is still a valid XML Schema.

Table 2 Example of resolution of nested occurrences of tag <choice>

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>

Among the XML Schema constructs, a special handling is reserved to <all>, <group>, and <attributeGroup>, for which a preprocessing phase is applied. Considering <all>, its meaning is that its child elements are to be taken altogether, and can appear in any order. Thus for testing purposes this *category* requires the validation of two different capabilities: the ability of the unit receiving as an input a valid instance to correctly accept each of the data listed in the <all> body; in addition, the capability of the unit to read those values whatever is the order in which they appear. Since there exists no criterion for privileging an ordering over another, or, said differently, all of the combinations have the same possibility of revealing faults, we consider only one of them, taken at random, as a representative. This contributes to reducing the number of test cases obtained with the application of CP. For example, from a XML Schema like:

```
<element name="pers_data">
  <complexType>
    <all>
      <element name="name" …/>
      <element name="surname" …/>
    </all>
  </complexType>
</element>
```

after processing of the tag <all> we could obtain:

```
<element name=”pers_data”>
  <complexType>
    <sequence>
      <element name=”surname” …/>
      <element name=“name” …/>
    </sequence>
  </complexType>
</element>
```

Considering <group>, and similarly <attributeGroup>, since they are used to define a group of elements (attributes) to be used in <complexType> definitions, we simply copy the body[ii] of each <group> and <attributeGroup> element in each <complexType> which refers them. This of course does not contribute to the definition of the test instances, but simplifies their successive automatic derivation.

Summarizing, after this preprocessing phase, the resulting sub-XML Schemas will be rewritten without the presence of <choice>, <all>, <group>, and <attributeGroup> tags. For each obtained sub-schema/functional unit, then we derive the *categories* as corresponding to each of the remaining XML Schema constructs among those reported in Table 1, i.e.: <simpleType>, <complexType>, <attribute>, <sequence>, <simpleContent>, <complexContent>.

For each of these *categories*, a quite standard processing is applied. We identify the *choices*, i.e., the relevant values that element can take, by analyzing the type of data defined for the corresponding XML Schema construct. Then, according to the CP method, we annotate each *choice* with its *constraints*, which in XPT are represented by the restrictions and/or facets listed for each datatype in the XML Schema definition.

Once identified the exact range in which the elements can vary, we randomly select a certain number of values from the specified subset of input. The number of *choices* to be considered is established taking into due consideration: the minimum and maximum values declared for each *category*; and also boundary value analysis, according to which selecting the test values near to the edges of input subdomains is more effective at finding failures .

More precisely, the test instances generation starts considering all the <simpleType> (and similarly <attribute>) elements. For each of them the *choices* are determined by the analysis of their type, that could be for instance integer, string, date, boolean and so on. Consider the example below:

```
<schema>
  <element name="shiporder">
    <complexType>
      <sequence>
        <element name="orderperson" type="xs:string" />
        <element name="shipto">
          <complexType>
            <sequence>
              <element name="name" type="xs:string" />
              <element name="address" type="xs:string" minOccurs="1" />
              <element name="city" type="xs:string" />
              <element name="country" type="xs:string" />
            </sequence>
          </complexType>
        </element>
        <element name="quantity" type="integer" minOccurs="2" />
```

[ii] This is done after we have processed possible <choice> and <all> elements that could appear inside the <group> body.

```
          <element name="note" type="string" maxOccurs="unbounded" />
        </sequence>
        <attribute name="orderid" type="xs:string" use="required" />
      </complexType>
    </element>
  </schema>
```

We first consider the elements named: “orderperson”, “name”, “address”, “city”, “country”, “quantity”, “note”, “ordered” and for each of them we analyze the associated type. For instance for “orderperson” the possible *choices* will be selected in the string range.

In case of <simpleType> other restrictions can be used for bounding the range of possible inputs, such as the *minInclusive* and *maxInclusive* declared values, which specify the lower and upper bounds respectively for numeric values, *minLength* and *maxLength* which define the minimum and maximum number respectively of characters or list items allowed, or *enumeration* which defines a list of acceptable that must be considered. Then for deciding the number of *choices* we analyze the other *constraints* declared. For instance the element “quantity” has minOccurs="2", thus for this element we select two *choices*.

Once the *choices* for <simpleType> and <attribute> elements are established, by using the tree structure of XML Schema, we combine them together for deriving the XML test instances. Let us explain the approach with an example. Considering for instance the *category* <complexType> labelled “shipto”, its *choices* are obtained as all possible combinations of the *choices* of its composing elements, in the example combined according to the rule for <sequence>. Hence the *choices* of “shipto” are all the possible combinations of the *choices* associated with the “name”, “address”, ”city”, “country” elements.

Thus supposing that the *choices* for name and address are {Mario, Mary} {Via Nazionale , Oxford Street}, {Roma, London}, {Italy, UK} two pieces of the resulting instances could be:

```
<shipto>
  <name> Mario </name>
  <address>Via Nazionale</address>
  <city> Roma </city>
  <country> Italy </country>
</shipto>

<shipto>
  <name> Mario </name>
  <address> Oxford Street </address>
  <city> Roma </city>
  <country> UK </country>
</shipto>
```

From the latter example (Roma, UK), note that we cannot check the semantic validity of the generated instances.

Building Test Instances

We have seen so far how the Category Partition can be naturally applied to XML Schemas yielding an automated and controllable strategy for generating XML test instances. However following faithfully such method would result in a huge number of XML instances. In order to reduce the number of generated test instances, it is convenient to introduce some practical improvements. Test reduction can be driven by specific consideration on the XML technology and at the same time by due care of not eliminating important test instances.

The abstract scheme in Figure 1 can aid explaining the main steps of XPT and refers to the architecture of the implemented tool TAXI, which goes from XML Schemas to a set of XML Instances.

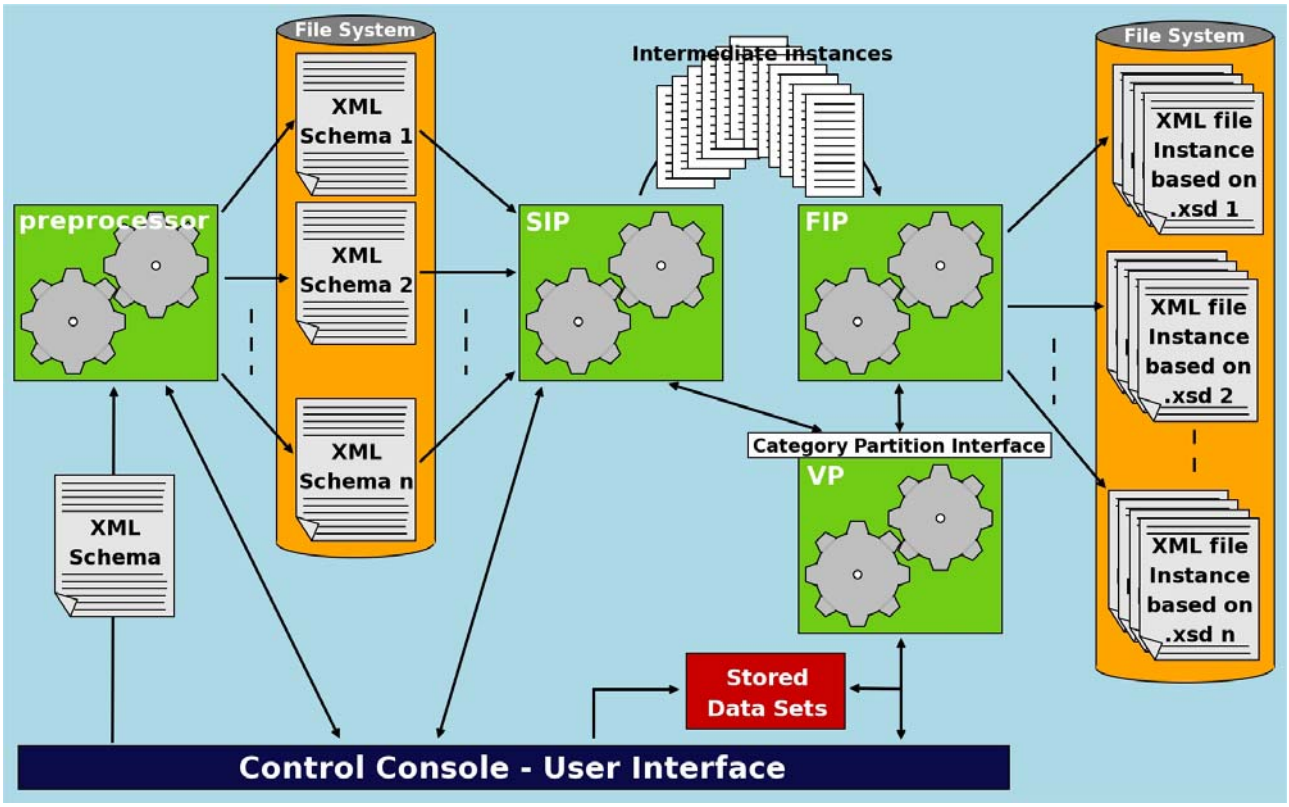

Figure 1 Main elements in the TAXI tool

TAXI has a User Interface (currently only a textual version exists, but we are working on a graphical version) from which the tester can control the generation process. The tool takes in input the XML Schema to start the generation from. The XML Schema is first passed to the preprocessor component. Scope of this component is solving tags `<choice>` and `<all>`. As we have seen this step will generate multiple sub-XML schemas, depending on the number of the `<choice>` constructs. The preprocessor also removes the `<group>` and `<attributeGroup>` tags expanding inline each occurrence of them.

Each obtained sub-Schema is stored in the file system in a separate directory from which the component SIP (Skeleton of Instances Producer) retrieves and analyses them one by one. Main task of SIP is to solve the issues related to occurrences of each element for which the condition *minOccurrences* ≠ *maxOccurrences* holds and then to produce intermediate files that reflect the skeleton of a set of final instances. In Table 3 an example is reported of two intermediate files generated from the Schema showed in the first left-hand side column of the same table.

Table 3 extract from an intermediate files example

Schema	Skeleton file for $n^{<B>}=2$	Skeleton file for $n^{<B>}=4$
… <element name="A"> <complexType> <sequence> <element name="B" minOccurs="2" maxOccurs="4" type="String"…> <element name="C" type="String"…> </sequence> </complexType> </element> …	… <A> <**B**><CNRISTInstr type="String"…/></**B**> <**B**><CNRISTInstr type="String"…/></**B**> <C><CNRISTInstr type="String"…/></C> </A> …	… <A> <**B**><CNRISTInstr type="String"…/></**B**> <**B**><CNRISTInstr type="String"…/></**B**> <**B**><CNRISTInstr type="String"…/></**B**> <**B**><CNRISTInstr type="String"…/></**B**> <C><CNRISTInstr type="String"…/></C> </A> …

The number of occurrences of each element to be used in the generation of the Skeleton files, is returned by the component labelled as VP (Values Provider). According to CP methodology, the number of skeletons to be produced results from the all possible combinations of the value assigned to each element.

Figure 2 can aid to understand the idea. On the top left-hand side of the figure a Schema is represented in which there are multiple elements with different values for *maxOccurrences* and *minOccurrences*. In particular for reducing the number of possible combinations, accordingly with the strategy of boundary condition adopted, only the border values *minOccurrences* and *maxOccurrences* will be considered, i.e., values 2, 4 for element "B" and 3, 4 for "C". Starting from these values the SIP component will derive a tree data structure in which each node is labelled with the name of the corresponding elements and each arc with the set of chosen values. Correspondingly the resulting valid couples of occurrences for elements "B" and "C" will be (2,3), (2,4), (4,3), (4,4).

Successively Figure 2 reports all the possible combinations of the couple listed above for a double sequence structure. Each column corresponds to a diverse valid XML instance. Finally in the bottom part of the figure the instance corresponding to the highlighted couple is shown explicitly. As it can be observed, the resulting number of skeleton instances is quite high even for this simple Schema, thus methods for reducing the possible combinations could be useful. To this purpose TAXI currently implements the criterion of considering each tuple at list once (it is possible that some tuples will be repeated not to leave an incomplete sequence). Thus considering the example of Figure 2 two skeletons will be derived: the first using the two couples (2,3) and (4,3) and the second the couples (4,4) and (2,4).

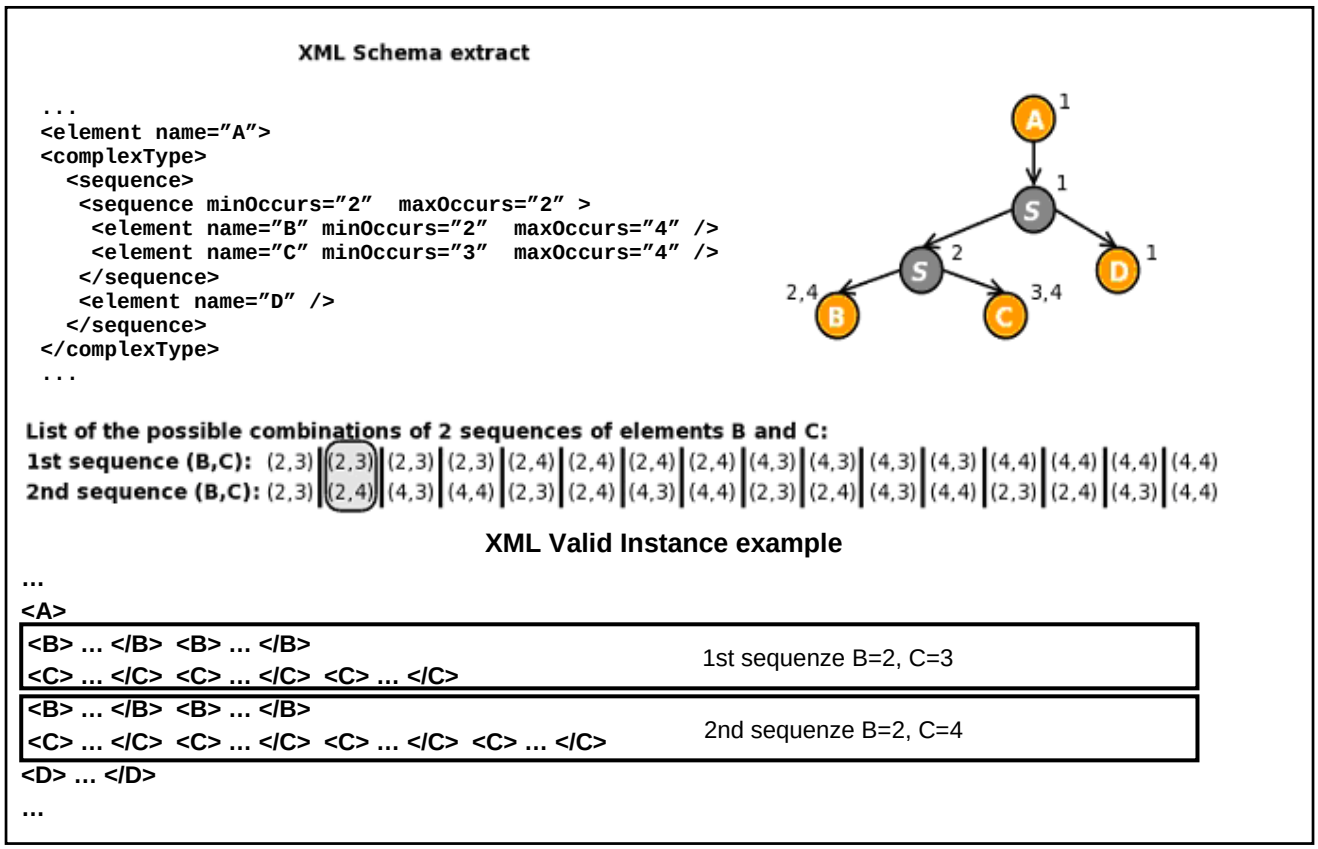

Figure 2 Strategy followed by the SIP component

The reduction steps we incorporate in XPT are conceived to avoid the repetition of similar tests. In general, starting from an XML formatted input, we actually test a component against two purposes. The first, more obvious, purpose concerns the validation of the component functional behaviour. Instead the second concerns the capability of correctly interpreting the data contained in the XML file, as specified by the corresponding XML Schema. For the latter purpose, our reduction heuristic relies on the hypothesis that two test instances providing the same input elements in a different ordering are equivalent. If this is not case, our reduction step could affect the failure detection capability.

The skeletons of instances so produced are finally analysed by the FIP (Final Instance Producer) component. Scope of this component is to analyse the instructions in the skeleton files and generate from each skeleton a set of test instances (by construction conforming to the original schema), using different values provided by the VP component.

This generation process is a resource consuming effort. The interaction between the SIP and FIP components in Figure 1 can be conceived as an instance of a Read-Write buffer that stores instances of Document Object Model (DOM)[6], each one representing an intermediate file. The XML Document Object Model (DOM) is a W3C standard method for accessing and manipulating XML documents. It stores in memory an XML instance as a tree structure that can be browsed in any direction. This solution makes it possible to parallelise SIP and FIP execution, and also avoid the storage of all the skeleton files on the file systems.

As said the selection of the values to be inserted in the final instances is done by component VP. This component contains the logic that permits to define the *choices* within a *category* and correspondingly to select values for the *choices*. The identification of the *choices* within a basic type generally depends on the nature of the data. For instance in a XML Schema a string could represent ZIP codes and the tester could be interested in verifying that the implementation is able to

recognize ZIP codes from all the fifty states of the USA. For this reason we could group within the same *choice* the ZIP code from the same state. The "Stored Data Sets" component in the bottom right-hand side of Figure 1 permits to store such kind of association between *choices* and elements in the schema. However when no *choices* have been specified for a particular element or attribute, by default `VP` will operate returning random values according to the *choices* defined for each basic type.
In some cases, such as to deal with exceptional test cases, the tester could want to insert some values by hand. This can be done passing in manual mode for some element (specified in the "Stored Data Sets" component).

XPT as a standard practice

Automated generation of test inputs that can systematically sample a program's behavior has been an aspiration of researchers and practitioners since the early days of the software testing discipline. In particular, input domain partitioning is acknowledged as an effective practical class of black-box test techniques, but the issue to solve has always been the adoption of rigorous yet practical input specifications amenable to mechanized parsing and sampling. Thus, techniques and prototype tools have been proposed, variously based on specific formal or semiformal notations for defining the input domain, but hardly they can be adopted inside existing development processes different from those for which these test methods have been conceived.

With the advent of XML and XML Schema as the standard format and grammar, respectively, for data representation and exchange, we believe that we have eventually reached the point at which automated test generation can be routinely and effectively done for the units composing a complex system as part of a standard compositional approach.

The introduced XPT method is the typical Columbus's egg: once explained, it seems quite obvious. Indeed, starting from XML Schema and applying to it the Category Partition test methodology, as we have shown in this paper, so to automatically derive test inputs in the form of compliant XML instances, really seems a captivating idea.

We have illustrated the main steps of the XPT approach, and the main components of a proof-of-concept tool TAXI implementing it. The tool is undergoing development and there are many technical details to be solved yet. However, we really see this approach as a promising and viable solution towards the adoption of black box partition testing as a routine procedure at the unit test stage.

References

1. Richardson D, J. , Clarke L., A. Partition Analysis: A Method Combining Testing and Verification. *IEEE Trans. Software Eng.* 11(12) pp. 1477-1490 (1985)
2. Szyperski c. Gruntz D., Murer S. *Component Software –Beyond Object-Oriented Programming* Addison-Wesley, 2nd edition, 2002
3. Extensible Markup Language (XML); http://www.w3.org/XML/
4. Bradley N. *The XML Schema Companion* Person Education, 2004
5. XML-XIG; http://xml-xig.sourceforge.net/
6. Document Object Model (DOM); http://www.w3.org/DOM/

The Category Partition method

Introduced in the late 80's, the CP provides a systematic, semi-automated method for test data derivation, starting from analysis of specifications until production of the test scripts, through a precisely defined series of steps:

1- Analyze specs and identify the *functional units*: these are the parts of the system that can be tested separately (for instance, according to design decomposition)
2- Partition the functional specifications of the unit into *categories*: i.e., for each unit, identify the environment conditions (the required system properties for a certain functional unit) and the parameters (the explicit inputs for the unit) that are relevant for testing purposes (hints: look for interface arguments, global variables that influence the output in significant way, …).
3- Partition the categories into *choices*: for each category, the choices represent significant (from the tester's viewpoint) values that it can take: clearly choice selection greatly influences testing effectiveness.
4- Determine *constraints* among choices. Since a suite of test cases is in principle obtained by taking *all the possible combinations of choices for all the categories*, in CP the choices can be annotated with *constraints*, in order to prevent the construction of redundant, not meaningful or even contradictory combinations of choices. Constraints can be of two types: *i*) either properties or *ii*) special conditions, such as the markings "error" and "single" referring to erroneous or special conditions, respectively, that need not to be combined with all possible choices.
5- Derive the *test specification*: categories, choices and constraints form a Test Specification, suitable for automatic processing. It is not yet a list of test cases, but contains all the necessary information for instantiating them by unfolding the constraints.
6- Derive and evaluate the *test frames*. From the test specification, a set of test frames is derived (automatic generation). Test frames are obtained by taking every allowable combination of categories, choices and constraints. After this step, test frames are subject to tester's evaluation, and if needed the Test Specification is modified (e.g., to reduce the number of tests).
7- Generate the *test scripts*: finally, test scripts are derived from the test frames; a test script is one or a sequence of test cases that can be executed

T. J. Ostrand and M. J. Balcer, The Category-Partition Method for Specifying and Generating Functional Tests, Communications of ACM, Vol. 31, No. 6, 1988.