Choreography Scenario-based Test Data Generation

Kai Ma,Jin Wang,Hongli Yang
College of Computer Science
Beijing University of Technology
Beijing, China
{mk,wj1990,yhl}@bjut.edu.cn

Jun Yan,Jian Zhang
Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
{Jun Yan,zj}@ios.ac.cn

Shengchao Qin
School of Computing
Teesside University, UK
{S.Qin}@tees.ac.uk

Abstract

Web service choreography specifies a sequence of interactions among multiple services. How to test if a Web service conforms with given choreography specification is a challenging question. It is important to generate test data (i.e. XML instance) based on the choreography. Since choreography scenarios describe expected interactions among multiple participants, it is possible to generate test data based on those scenarios. This paper presents a set of test data generating rules and algorithms based on refined type trees, which are obtained from choreography scenario and corresponding XML Schema type document. We have built a prototype tool to support automatic test data generation and illustrate the process of generating XML instances via a purchase order choreography scenario example.

Keywords: Web Service Choreography; Scenario; XML Schema; Test Data Generation

1. Introduction

A choreography specification identifies allowable ordering of message exchanges in a distributed system, and has been considered a scalable and decentralized solution for composing Web services. The choreography conformance problem is identifying if a set of given services adhere to a given choreography specification. It has been an important research area in service oriented computing in the last several years. For testing conformance of choreography, It is important to generate test data from choreography specification.

A choreography scenario represents sequence of interactions among participants, it involves not only a sequence of interactions, but also message variable type declarations, which are defined in a referred XML Schema document. Due to the indicators such as $choice$, $minOccurs$ and $maxOccurs$, used in XML Schema type definition, a message variable type in a choreography scenario may be too complex to be used directly for generating XML instances.

In a previous preliminary exploration [1], we partitioned XML Schema type into subtypes, and presented the definition of XML Schema type tree. Moreover, we use combinatorial tool $Cascade$ [2] to obtain refined type trees. In this paper, based on refined type trees, rules and algorithms are presented for generating XML instances. We use a purchase order choreography scenario example to show the generation process of XML instances. A tool has been developed for supporting automatic implementation of generation.

Supported by open foundation of State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences (No. SYSKF1008), and Subject and Postgraduate Education Construction Project of Beijing Municipal Commission of Education.

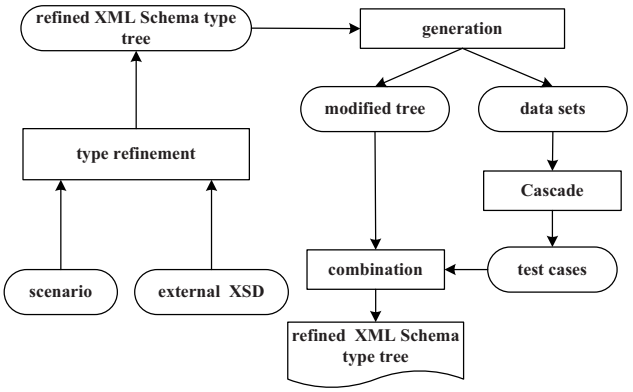

Fig. 1. Test Data Generation Process Based on Choreography Scenario

2. Test Data Generation Process Based on Choreography Scenario

2.1. Choreography Scenarios

A choreography scenario is a 4-tuples $\langle R, I, V, A\rangle$, where R is a finite set of role declarations; I is a finite set of information type, which is defined in an external XML Schema document; V is a finite set of variable declarations, which map each variable with its type; A is a sequence of interations.

In the previous studies [1], we have introduced an example of purchase order choreography scenario and XML Schema type definition for purchase order, which will be used again for demonstrating the process of generation test data.

978-1-4799-5029-4/14 $31.00 © 2014 IEEE
DOI 10.1109/TASE.2014.25

2.2. Test Data Generation Process

The figure 1 shows the process of choreography scenario-based test data generation, which are described as following

- For an input choreography scenario and an external XML Schema document, the module $type\ refinement$ builds a set of refined type trees.
- The module $generation$ modifies a refined type tree and generate $data\ sets$, which can further be reduced by $Cascade$.
- The reduced $test\ cases$ will be combined with $modified\ tree$ and finally generate XML instances.

3. Generating Test Data from Refined Type Tree

3.1. Refined Type Tree

A refined type tree T is a 4-tuples:

$$T = \langle N, r, C, E \rangle$$

where:

- N is a finite set of element nodes and control nodes such as $sequence$ and $choice$. Particularly, All the leaf nodes are element nodes with built-in types, which may have a set of facet constraints.
- r is a root node.
- C is a finite set of indicator constraints. Each constraint in C may take either one of the basic constrains below or the join of them:
  - $\langle occurrenceTime, n \rangle$ means the node will occur for n times, here n is nature number and default value is 1.
  - $\langle choice_i, N \rangle$ means the node N will be the $i\ th$ child of its father node.

  In this paper, the indicator that we mainly focus on are $minOccurs$, $maxOccurs$ and $Choice$.
- E is a finite set of edges. An edge can be expressed as $e(m, c, n)$, where $c \in C, m \in N \cup \{r\}$ and $n \in N$.

A refined purchase order type tree is shown in figure 2, which is introduced in the previous studies. The numbers on the edges of the tree show the occurrence time of its child node, and explicit the information of XML Schema indicators under refinement.

3.2. Generating Rules

We give the data generation rules for built-in types with facet restriction. According to XML Schema definition, we consider four kinds of facets for built-in types xs:int and xs:$string$. They are: $maxInclusive/minInclusive$, $enumeration$, $maxLength/minLength$ and $pattern$.

For each facet, we can generate both valid and invalid data as follows. Here the function $randomInt(n1, n2)$ generates a random integer, which is less or equal to $n2$ and larger or equal to $n1$, and $randomStr(n)$ generates a random string with length n.

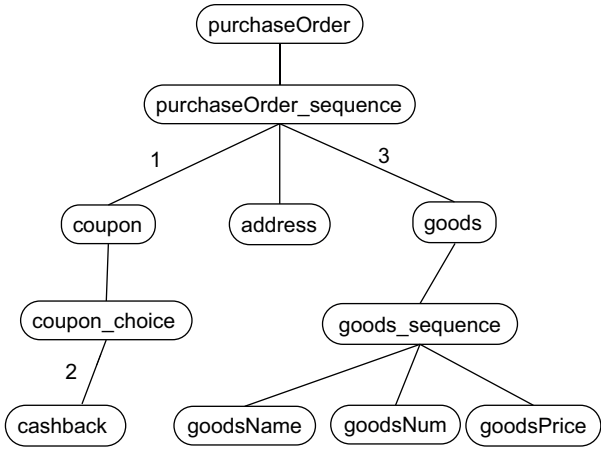

Fig. 2. A Refined Purchase Order Type Tree

- Rule 1: $maxInclusive/minInclusive$
  valid data set has:
  - $minInclusion$
  - $minInclusion$+1
  - $randomInt(minInclusion, maxInclusion)$
  - $maxInclusion - 1$
  - $maxInclusion$

  invalid data set has:
  - $minInclusion - 1$
  - $maxInclusion + 1$
- Rule 2: $enumeration$
  valid data are the strings inside of enumeration set; invalid data are strings generated randomly and not inside of enumeration set.
- Rule 3: $maxLength/minLength$
  valid data set has
  - $randomStr(minLength)$
  - $randomStr(maxLength)$
  - $randomStr(randomInt(minLength, maxLength))$
  - $randomStr(minLength + 1)$
  - $randomStr(maxLength - 1)$

  invalid data set has
  - $randomStr(minLength - 1)$
  - $randomStr(maxLength + 1)$
- Rule 4: $pattern$
  valid data are strings fitting the pattern; invalid data are strings that do not fit the pattern. The tool[3] is used for generating matched strings.

Particularly, for built-in types $xs : string$ and $xs : int$, we simply use function $randomInt$ and $randomStr$ to generate valid data and invalid data as shown in what follows.

- Rule 5: $xs : string$
  valid data set is:

– $\{randomStr(randomInt(1, 10))\}$

invalid data set is:

– $\{randomInt(0, 10000)\}$

- Rule 6: $xs : int$

valid data set is:

– $\{randomInt(0, 10000)\}$

invalid data set is:

– $\{randomStr(randomInt(1, 10))\}$

In the following, some examples of XML Schema definition with facet restriction are presented. Using the generation rules, the test data set are given.

```
<xs: element name="L">
<xs: simpleType>
  <xs:restriction base="xs:integer">
   <xs:minIclusive value="0">
   <xs:maxIclusive value="100">
  </xs:restriction>
</xs: simpleType>
</xs:element>
```

Using the Rule 1, the valid data set of element L is $\{0, 1, 37, 99, 100\}$, and invalid data set is $\{-1, 101\}$.

```
<xs: element name="M">
<xs: simpleType>
  <xs:restriction base="xs:string">
   <xs:enumeration value="Audi">
   <xs:enumeration value="BMW">
   <xs:enumeration value="Ford">
  </xs:restriction>
</xs: simpleType>
</xs:element>
```

Using the Rule 2, the valid data set of element M is $\{"Audi","BMW","Ford"\}$, and invalid data set has strings outside of valid data set.

```
<xs: element name="N">
<xs: simpleType>
  <xs:restriction base="xs:string">
   <xs:minLength value="5">
   <xs:maxLength value="8">
  </xs:restriction>
</xs: simpleType>
</xs:element>
```

Using the Rule 3, the valid data set of element N is $\{"aabbc","aabbccdd","aabbcc","aabbccd"\}$, and invalid data set is $\{"bbbb","bbbbbbbb"\}$.

```
<xs: element name="O">
<xs: simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="[a-z]">
  </xs:restriction>
</xs: simpleType>
</xs:element>
```

Using the Rule 4, the valid data of element O is matched with pattern "$[a-z]$", and invalid data is not matched with the pattern.

3.3. Generating Algorithms

In this section, we introduce the process and algorithms for generating XML instances.

The algorithm $generation$ takes a root n of a refined XML Schema type tree as input, and outputs a modified tree and data sets, which are corresponding with the values of leaf nodes of the tree.

Algorithm: $generation$
Input: refined XML Schema type tree T'=$\langle N, r, C, E\rangle$
Output: modified tree and data sets V
1 $n = T'.r$
2 $V = \emptyset$
3 $derive(n)$
4 **return** T' *and* V

Algorithm: $derive(n)$
1 **if** n *is sequence node*
2 $\quad n.father.childrenlist.add(n.childrenlist)$
3 $\quad remove\ n\ from\ n.father.childrenlist$
4 $\quad$ **return**
5 **else if** n *is choice node*
6 $\quad$ **if** $getOccurrenceTime(n) == 1$
7 $\quad\quad n.father.childrenlist.add(n.child)$
8 $\quad$ **else**
9 $\quad\quad$ **for** $i = 0; i < getOccurrenceTime(n); ++i$
10 $\quad\quad\quad addSiblingByOrderIndex(n,\ i))$
11 $\quad remove\ n\ from\ n.father.childrenlist$
12 $\quad$ **return**
13 **if** $(t = getOccurrenceTime(n)) > 1$
14 $\quad$ **for** $i = 0; i < t - 1; ++i$
15 $\quad\quad n.father.childrenlist.add(n)$
16 **if** n *is leaf node*
17 $\quad generate\ n's\ Dataset\ to\ V$
18 $\quad$ **return**
19 **foreach** $child\ in\ n.childrenlist$
20 $\quad derive(child)$

The auxiliary algorithm $getOccurrenceTime(n)$ takes node n as input, and returns the occurrence times of node n. The algorithm $addSiblingByOrderIndex(n,\ i))$ takes node n and its ith child as input, and returns updated node n, whose ith child is added to its sibling list.

Algorithm: $getOccurrenceTime(n)$
1 **foreach** $c\ in\ e(n.father, C, n).C$
2 $\quad$ **if** $c\ is\ \langle occurrenceTime, t\rangle$
3 $\quad\quad$ **return** t

Algorithm:$addSiblingByOrderIndex(n,\ i))$
1 **foreach** $c\ in\ e(n.father, C, n).C$
2 $\quad$ **if** $c\ is\ \langle choice\_i, t\rangle$
3 $\quad\quad n.father.childrenlist.add(t)$

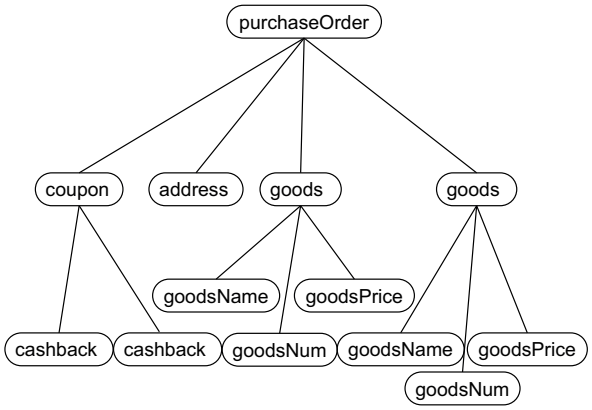

Fig. 3. Modified Tree of Purchased Order Type

Here are the main steps of the generation process:

Step 1: Traverse refined type tree to eliminate control nodes and indicator constraints under the following rules:

1) For sequence node, move the children nodes of it to be its sibling and then remove the sequence node.

2) For choice node that has only one child, move the child to be sibling of it and remove the node.

3) For choice node that has more than one child, firstly, clone its children node one by one following the document order, and add them as its sibling, then delete the choice node and its children. The clone of a node here means deep copy of its children's information as well as its own.

4) For non-control node, if occurrence times of it is n, which is more than once, clone it for $n-1$ times and add them to be its own sibling.

Step 2: When reaching a leaf node during traversal, identify the node and generate data set for it by the rules in Section 3.2.

Step 3: After traversing and modification, the input tree has been changed as an XML instance tree, which forms valid and invalid XML instances after combining with generated data sets.

The Figure 3 is the modified tree, which is a intermediate model of $generation$ algorithm. We can see it as the structure of XML instance.

There are data sets generated together with the modified tree in Figure 3. Table 1 shows the generated valid data sets.

TABLE 1. Valide data Sets Generated

No.	node name	valid data set
0	cashback	{1,2,18,99,100}
1	cashback	{1,2,5,99,100}
2	address	{"bjut001"}
3	goodsName	{"aabbc","aabbccdd","aabbcc","aabbccd"}
4	goodsNum	{1}
5	goodsPrice	{0,1,88,99,100}
6	goodsName	{"aabbc","aabbccdd","aabbcc","aabbccd"}
7	goodsNum	{1}
8	goodsPrice	{0,1,50,99,100}

By combining modified tree with generated data sets, we can obtain a set of XML instances. We tried several strategies for combination of them. In the easy way, we can randomly pick one value from corresponding set for each node, and it may lose too much detective ability along with low coverage. However, highly coverage satisfaction by full combination of all sets will result in cost problem. The combinatorial approach is the most suitable method for this. With the help of $Cascade$ tool, we get 217 test cases instead of $5*5*4*5*4*5 = 10000$ test cases without combinatorial approach. For each test case, we finally combine the data value with corresponding modified tree, and get a set of XML instances. The following is one of XML instance for $purchaseOrder$ type defined in [1].

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder>
  <coupon>
   <cashback>1</cashback>
  <cashback>1</cashback>
  </coupon>
  <address>bjut001</address>
  <goods>
   <goodsName>aabbc</goodsName>
   <goodsNum>1</goodsNum>
   <goodsPrice>99</goodsPrice>
  </goods>
  <goods>
   <goodsName>aabbccdd</goodsName>
   <goodsNum>1</goodsNum>
   <goodsPrice>50</goodsPrice>
  </goods>
</purchaseOrder>
```

4. Conclusion and Future Work

This paper presents our work about test data generation based on choreography scenarios. The main contribution of this paper includes:

- presents a choreography scenario based approach for generating test data, which can be used for testing conformance between choreography and its implemented Web services.
- presents algorithms for generating test data from refined type tree.
- uses combinatorial testing approach to reduce test cases.

For the future work, we intend to continue the development of our tool, and design experiments for evaluating our approach.

References

[1] H. Yang, K. Ma, C. Deng, H. Liao, J. Yan, and J. Zhang, "Towards conformance testing of choreography based on scenario." in *TASE*. IEEE, 2013, pp. 59–62. [Online]. Available: http://dblp.uni-trier.de/db/conf/tase/tase2013.html#YangMDLYZ13

[2] "Cascade: CAS Covering Array DEsigner," http://lcs.ios.ac.cn/~zhangzq/ct_toolkit.html.

[3] "Regxstring," http://regxstring.googlecode.com/files/regxstring_0.1.tar.gz.