HKU CS Tech Report TR-2009-15

Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundation of Software Engineering (*ESEC 2009/FSE-17*), ACM Press, New York, NY (2009)

Data Flow Testing of Service Choreography*†

Lijun Mei
The University of Hong Kong
Pokfulam, Hong Kong
ljmei@cs.hku.hk

W. K. Chan‡
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

ABSTRACT
Service computing has increasingly been adopted by the industry, developing business applications by means of orchestration and choreography. Choreography specifies how services collaborate with one another by defining, say, the message exchange, rather than via the process flow as in the case of orchestration. Messages sent from one service to another may require the use of different XPaths to manipulate or extract message contents. Mismatches in XML manipulations through XPaths (such as to relate incoming and outgoing messages in choreography specifications) may result in failures. In this paper, we propose to associate XPath Rewriting Graphs (XRGs), a structure that relates XPath and XML schema, with actions of choreography applications that are skeletally modeled as labeled transition systems. We develop the notion of XRG patterns to capture how different XRGs are related even though they may refer to different XML schemas or their tags. By applying XRG patterns, we successfully identify new data flow associations in choreography applications and develop new data flow testing criteria. Finally, we report an empirical case study that evaluates our techniques. The result shows our techniques are promising in detecting failures in choreography applications.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging—Testing tools; D.2.8 [**Software Engineering**]: Metrics—Product metrics

* © ACM (2009). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundation of Software Engineering* (*ESEC 2009/FSE-17*), ACM Press, New York, NY (2009). http://doi.acm.org/10.1145/1595696.1595720.

† This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 717506 and 717308).

‡ All correspondence should be addressed to Dr. W. K. Chan at Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. Tel: (+852) 2788 9684. Fax: (+852) 2788 8614. Email: wkchan@cs.cityu.edu.hk.

General Terms: Measurement, Reliability, Verification

Keywords: service composition, software testing, choreography, orchestration, web services, data flow testing.

1. INTRODUCTION
Software built on top of the service-oriented architecture (SOA) is increasingly popular [4][14][27], and yet testing such software remains tedious and ill-understood. In these applications, primary components are typically known as *services*. A set of such coordinated components are called a *service composition*. To synthesize a service composition, one may coordinate services via two general strategies [13][17], namely *orchestration* [14] and *choreography* [18]. Testing, analysis, and formal verification techniques should address the challenges presented by these strategies.

Orchestration specifies the sequences of process actions of individual services [13]. Through collaboration (e.g., parallel composition of processes in process algebra [27]), such sequences of process actions of one service can coordinate, albeit implicitly, with the sequences of process actions of other services in the same application. Techniques to test service orchestration have been proposed. For instance, Li et al. [10] model every business process as a composition model, and propose a unit test framework to verify them. Mei et al. [14] examine the verification problem from the perspective of loose coupling, formulate how a service orchestration manipulates XML through XPath, and develop a class of data flow techniques. Their techniques not only reveal the coverage properties of XML via XPath and the associated XML schema (or DTD), but also direct test efforts to cover [14][15] the derived requirements.

Choreography (see [23][25], for example) defines the rules of interactions, common message types, and exchange agreements among a set of services [18]. For instance, instead of using implicit collaboration sequences embedded in the parallel composition of processes [27], choreography explicitly defines the interactions among services by such means as *Message Sequence Charts* (*MSCs*) [19][20], UML sequence diagrams, or the Web Services Choreography Description Language (WS-CDL) [22]. An application developed in this way is called a *choreography application*.

Existing research on service choreography focuses on service modeling [3], process flow modeling [2], synthesizing of models [21], verification using finite-state automata [7], and violation detection of properties such as atomicity and resource constraints [6][27]. Not much public literature highlights the testing challenges. Although XML has been considered in previous research [9] and the impact of XPath on services has been studied [14], surprisingly, as we are going to present in Section 3, the mismatch among service interfaces in the testing of service choreography has not been addressed in the literature.

Let us consider an example of a choreography application based on a WS-CDL specification. For each service, the specification explicitly defines the accepted types for both incoming and outgoing messages. It also defines how the ports of one service are individually linked to other services in the same collaboration. It may further define its own variables to ease the coordination task. For instance, it may specify XPath selections, such as X_2 and Y_2 in Figure 1, so that the service choreography can select parts of the XML data kept by a particular variable to be sent to (or received from) a particular service. Of course, the latter service may use its own XPath selection (defined, say, in its own WSDL [24] document) to select its required contents. Thus, to send a message m_1 from a service *A* to a service *B*, an XPath in the WS-CDL specification may only select and forward a portion, say m_2, of the message m_1. To send m_2 to *B*, another WS-CDL XPath may be used to restrict the contents to be seen by *B*, which essentially sends a further portion, say m_3, of m_2 to *B*. The service *B* may use its own XPath (defined, for instance, in *B*'s WSDL document) to pick the required contents from m_3. In such a typical WS-CDL scenario where services are loosely coupled, three XPaths are used. Even though we force these three XPaths to conform to the same XML schema, if they incompatibly refer to disjoint fragments of m_1, the required content originally kept at m_1 may not be retrievable by *B*, regardless of whether m_3 has reached *B*. In more complicated scenarios, to interpret the same XML message, a sender may refer to the sender's XML schema, a receiver may refers to the receiver's XML schema, and the WS-CDL specification may also have its own XML schema. Thus, testing techniques should address how different XML schemas in the same choreography application may interpret the same message by the same or different services in different collaboration steps.

Simplifying a message exchange as a label (such as modeling the exchange as a pair of sent/received messages without data in an MSC) would be ineffective in pinpointing where the testing effort should be spent. As such, owing to the adoption of XPath selection to manipulate the contents in incoming or outgoing messages, the artifacts XPath, WSDL, and XML should be explicitly addressed in the choreography application model to supports testing, analysis, and formal verification.

In this paper, we propose to extend Labeled Transition Systems (LTS) [20][21] to model the interactions among services in a choreography application. In LTS, a label represents an action. A test trace is a sequence of labels that starting from the initial state. It represents the sequences of labels covered by the execution of a test case. Checking the sequence of actions in such a trace can help detect negative scenarios [21] or other static properties. However, an action cannot help specify concrete content selection, which involves concrete data. XPath [26] and WSDL [24] are crucial to the extraction of desirable contents from XML messages [14], and yet they have been abstracted out in the standard LTS model. We extend LTS by incorporating *XPath Rewriting Graphs* (*XRGs*) [14], which captures the relationships between XPath queries and WSDL documents. Specifically, we associate each XPath query in a WSDL or WS-CDL document with the action that may use it. By doing so, any number of XRGs (possibly zero) can be annotated in an LTS. To cater for multiple XML schemas interpreting the same XML messages, we develop the notion of XRG patterns. We model each legitimate interpretation of the same XML message by a pair of service endpoints as a matching of tags between a pair of corresponding XRG patterns. We develop an automated strategy to generate XRG patterns, and select such pairs of patterns that relates matching tags. Since different XRGs are linked up by chains of XRG patterns, we successfully develop a new family of data flow testing criteria to support the testing of service choreography.

The main contribution of this paper is fourfold: (i) We propose a model to capture the interactions in service choreography. (ii) We develop the notation of XRG patterns. (iii) Based on our model and XRG patterns, we identify a new set of def-use associations in service choreography, and further propose a new family of data flow testing criteria. (iv) Finally, to the best of our knowledge, we provide the first case study on data flow testing for choreography applications. The experimental result shows that our approach can be more effective than orchestration testing and random testing in revealing failures in a choreography application.

The rest of this paper is organized as follows: Section 2 presents the preliminaries that lay the foundations of our approach. Section 3 gives a motivating example and outlines the testing challenges of choreography applications. Section 4 presents our effort to model choreography applications, and introduces our data flow testing criteria to measure the comprehensiveness of test sets. Section 5 reports an experimental case study, followed by discussions and related work in Section 6 and Section 7, respectively. Finally, Section 8 concludes the paper and proposes future work.

2. PRELIMINARIES

This section introduces the foundations of our work.

2.1 Service Choreography

In service orchestration, a service can be implemented in different programming languages (such as Java, Python, or C#), its interface is publicly exposed as a WSDL document, and the orchestration can be implemented as a WS-BPEL program [15]. Figure 1 depicts a scenario in which an orchestration services *A* collaborates with a choreography service *B*. Each of the services exposes its WSDL documentation, which specifies the XPath for selecting the contents of an XML message (as shown in the bottom of the figure). For these two services to collaborate, service choreography (written, say, in WS-CDL [22]) can be developed. Choreography can be used to specify the interoperability and interactions between multiple services in an application [22]. A WS-CDL specification not only specifies how different ports of individual services are related, but also defines its own variables (such as X_1 and Y_1 in the figure) and the corresponding XML schemas.

WS-CDL is a popular specification language in service choreography. In general, a service can be composed through an orchestration language (such as WS-BPEL) or a choreography language (such as WS-CDL). In this paper, we follow [3] to model a service as a black-box component (see also [12]) via WS-CDL.

2.2 Label Transition Systems

Labeled transition systems (LTS) and unlabeled transition systems are two types of state transition systems [19]. Uchitel et al. [20][21] have applied LTS to portray implied scenarios among interacting components. For ease of presentation, we adapt their definitions into Definitions 1 and 2.

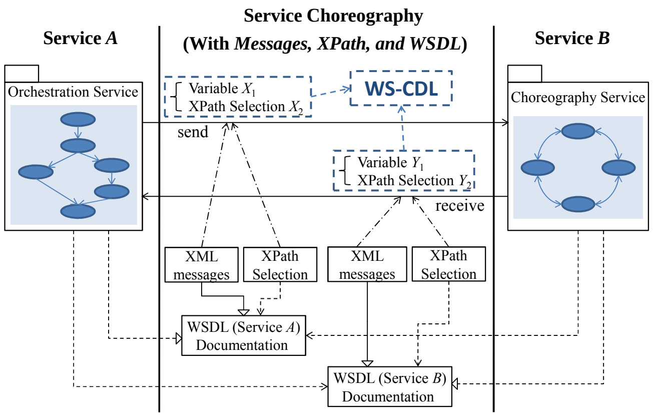


Figure 1. Key artifacts and their relationships in a typical service choreography application.

Definition 1 (Labeled Transition Systems). A finite *Labeled Transition System* (*LTS*) P is a structure $\langle S, L, \Delta, s\rangle$ such that S is a set of states; $L = \alpha(P) \cup \{\tau\}$ is a set of labels, where $\alpha(P)$ denotes the alphabet set of P, τ denotes the internal actions that cannot be observed by the environment of P; $\Delta \subseteq (S \times L \times S)$ is a set of transitions to link a state to another state; and $s \in S$ is the initial state of P.

Definition 2 (LTS Test Trace). Let $P = \langle S, L, \Delta, s\rangle$ be an LTS. A sequence $e = \langle s_1, l_1, s_2, l_2, \ldots\rangle$ is an execution of P if $s_1 = s$ and $\langle s_i, l_i, s_{i+1}\rangle \in \Delta$ for $0 < i < |e|$. A word $l = \langle l_1, l_2, \ldots\rangle$ over the alphabet set $\alpha(P)$ is an *LTS test trace* of P if there is an execution e of P such that $l = e|_{\alpha(P)}$.

We choose LTS as the skeleton of our model for choreography applications. Our considerations are as follows: (a) LTS is a formal model to represent message interactions for scenario-based specifications such as message sequence charts. (b) In this paper, we focus on choreography applications, and hence we can simply use τ to model internal actions of individual services. As such, our approach can generally apply to test choreography applications (with or without orchestration information). (c) The trace definition on top of LTS facilitates us to define data flow entities for choreography applications.

2.3 XPath Query Model

In WS-CDL [22], XPath must be used as the language to specify expressions, queries, and conditional predicates. We adopt the syntax definition of a decidable fragment of XPath from [16], as shown in Figure 2. According to [16], the fragment provides representative XPath syntax and is sufficient as the basis for studying XPath. Mei et al. [14] have also used this fragment in developing testing techniques.

			Rule
n(x)	=	{y | (x, y)∈*EDGES*(t), *LABEL*(y) = n}	… 1
*(x)	=	{y | (x, y) ∈*EDGES*(t)}	… 2
.(x)	=	{x}	… 3
(q_1/q_2)(x)	=	{z | y∈q_1(x), z∈q_2(y)}	… 4
(q_1//q_2)(x)	=	{z|y∈q_1(x), (y, u)∈*EDGES**(t), z∈q_2(u)}	… 5
(q_1[q_2])(x)	=	{y | y∈q_1(x), q_2(y)≠∅}	… 6

Figure 2. Syntax of a decidable fragment of XPath.

Mei et al. [14] have proposed an *XPath Rewriting Graph* (*XRG*) to represent an XPath with a model (Ω) of XML documents. Here, we revisit XRGs to facilitate the description of our new techniques. An XRG is built on XPath syntactic constructs [16]. Their technique treats the definitions in Figure 2 as left-to-right rewriting rules and, through a series of rewriting [5], transforms an XPath into a normal form or a fixed point. The intermediate rewriting steps are also recorded in an XRG. Every two consecutive steps are linked in the graph. We revisit the definition of XRG [14] in Definition 3.

Definition 3 (XPath Rewriting Graph). An *XPath Rewriting Graph* (*XRG*) for an XPath query is a 5-tuple $\langle q, \Omega, N_x, E_x, V_x\rangle$ such that

- q is an XPath expression for the XPath query, and Ω is an XML schema that describes the XML document to be queried on.
- N_x is a set of rewriting and rewritten nodes identified by the algorithm *Compute_XRG*, and V_x is a set of *conceptual variables* defined at the nodes in N_x.
- E_x is a set of edges (s_c, s_n), each of which represents a transition from s_c to s_n, where s_c is a rewriting node and s_n is either a rewriting node or a rewritten node. All the edges are also computed by the algorithm *Compute_XRG*.

The algorithm *Compute_XRG* [14] used in Definition 3 takes an XPath expression q, the schema Ω of some XML document, and a set of currently located nodes X of Ω as parameters, and outputs the corresponding XRG. (We refer to each node in X as a *tag*. This data structure forms an explicit artifact to model different paths, conceptually defined in an XPath, on how to provide query results.) In the algorithm, X is first initialized as a singleton set containing the root of the schema [16]. The query q starts with this value of X to search for other nodes. An example of an XRG will be shown in Figure 7.

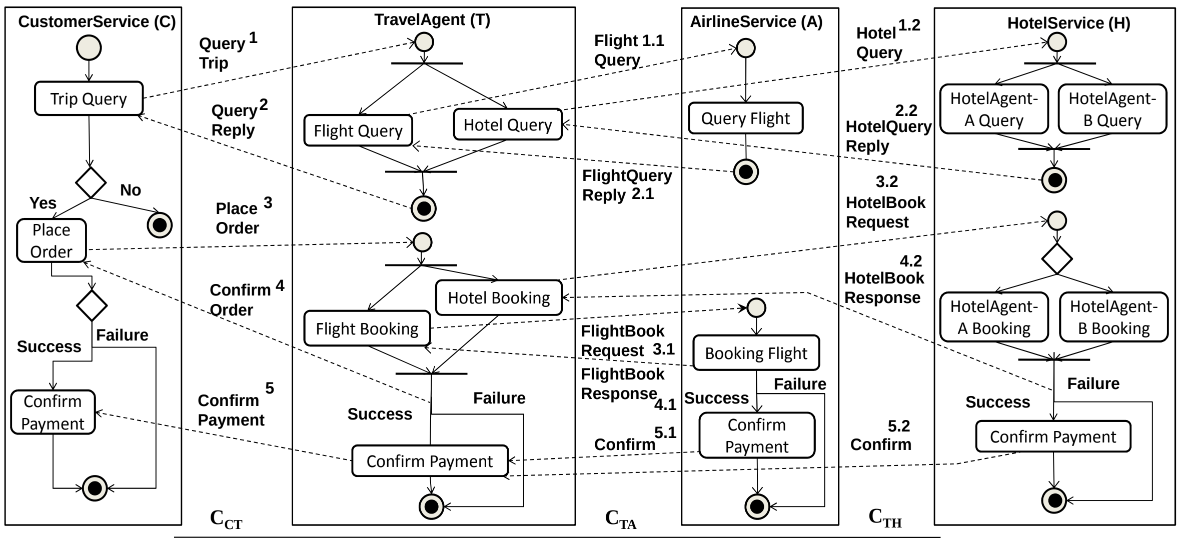
There are two types of XRG nodes [14]: *rewriting node* $\langle q, L^c, rule\rangle$ and *rewritten node* $\langle q, L^c, L^n, S\rangle$. Here, q is a query expression, L^c is the current set of nodes in Ω located by the previous query step, L^c is assigned to the root node of Ω, *rule* is the rewriting rule used to generate the sub-terms in the rewriting node, L^n denotes the set of nodes in Ω to be located by q starting from some node in L^c, and S is a set-theoretic representation of the result of q. In XRGs, we refer to the generation of the value of a variable ($\in V_x$) as *variable definition*, and the use of a variable provided by a preceding node as *variable usage*. Such variables are only conceptual in nature and are not program variables because they never appear in an implemented program. Hence, they are called *conceptual variables* [14]. By using inorder traversal of XRGs and dropping all rewriting nodes, *Compute_XRG* provides an explicit way to model different *conceptual paths* defined in an XPath for providing query results. In this paper, we use this algorithm to construct the XRGs in our model.

3. MOTIVATING EXAMPLE

This section uses a *TripHandling* project as a motivating example to introduce the testing challenges of choreography applications. The original source code is in WS-CDL [22] and is available from [25].

There are four roles in *TripHandling*: (a) *Customer Service*, which helps customers to place trip orders and process payments; (b) *Travel Agent*, which handles requests from service customers and schedules trips; (c) *Airline Service*, which handles flight schedule queries, seat booking, and online payments of flights; and (d) *Hotel Service*, which handles hotel queries, room booking, and online payments via two agents known as *HotelAgent-A* and *HotelAgent-B*. For instance, *Travel Agent* includes the following workflow steps: (i) *FlightBooking* books the flights for a trip. (ii) *HotelBooking* books the hotels for the trip. (iii) *Confirm-Payment* checks whether the corresponding charges (for successful flight and hotel reservations) have been paid by an online credit card service. (iv) If both *FlightBooking* and *HotelBooking* are successfully completed, the booking results will be sent to the customer. On the other hand, if either *FlightBooking* or *HotelBooking* encounters an error, the *TripHandling* application will display an error message and terminate. Descriptions of the other services are omitted owing to space reason.

The structure of the *TripHandling* application is shown in Figure 3(a). For ease of illustration, we follow [14][15] and use a UML activity diagram to depict the application. A service is portrayed as



(a) One Choreography Consisting of Three Sub-Parts (C_{CT}, C_{TF}, and C_{TH})

1
```
<xsd:complexType name="tripQuery">
   <xsd:element name= "departureDate" type="xsd:date"/>
   <xsd:element name="returnDate" type="xsd:date"/>
   <xsd:element name="fromCity" type="xsd:string"/>
   <xsd:element name="toCity" type="xsd:string"/>
   <xsd:element name="airlineName" type="xsd:string"/>
   <xsd:element name="hotelName" type="xsd:string"/>
</xsd:complexType>
```

2
```
<xsd:complexType name="tripQueryReply">
    ……
   <xsd:complexType name="tripQueryList"
                         type="xsd:tripQueryReply"/>
</xsd:complexType>
```

XML Schemas for C_{CT}

1.1
```
<xsd:complexType name="flightQuery">
   <xsd:element name= "departureDate" type="xsd:date"/>
   <xsd:element name="returnDate" type="xsd:date"/>
   <xsd:element name="fromCity" type="xsd:string"/>
   <xsd:element name="toCity" type="xsd:string"/>
   <xsd:element name="airlineName" type="xsd:string"/>
</xsd:complexType>
```

2.1
```
<xsd:complexType name="flightQueryReply">
    ……
    <xsd:element name="airlineName" type="xsd:string">
    <xsd:complexType name="airlineNameList"
                         type="xsd:airlinelist"/>
</xsd:complexType>
<xsd:complexType name="airlineList">
   <xsd:element name="airlineName" type="xsd:string"
                          minOccurs="0"/>
<xsd:complexType>
```

XML Schemas for C_{TF}

1.2
```
<xsd:complexType name="hotelQuery">
   <xsd:element name="departureDate" type="xsd:date"/>
   <xsd:element name="returnDate" type="xsd:date"/>
   <xsd:element name="city" type="xsd:string"/>
   <xsd:element name="hotelName" type="xsd:string"/>
</xsd:complexType>
```

2.2
```
<xsd:complexType name="hotelQueryReply">
    ……
    <xsd:element name="hotelName" type="xsd:string">
    <xsd:complexType name="hotelNameList"
                         type="xsd:hotellist"/>
</xsd:complexType>
<xsd:complexType name="hotelList">
   <xsd:element name="hotelName" type="xsd:string"
                   minOccurs="0"/>
<xsd:complexType>
```

XML Schemas for C_{TH}

(b) Typical XML schemas for XML messages defined in WSDL documents (for C, A, and H to support C_{CT}, C_{TF}, and C_{TH}, respectively).

1-a
```
<tripQuery>
   <departureDate>2009-5-18</departureDate>
   <returnDate>2009-5-24</ returnDate>
   <fromCity>HongKong</fromCity>
   <toCity>Vancouver</fromCity>
   <airlineName>UnitedAirline</airlineName>
   <hotelName>Westin</hotelName>
<tripQuery>
```

1-b
```
<tripQuery>
    ……
   <airlineName>-</airlineName>
   <hotelName>-</hotelName>
<tripQuery>
```

XML Messages for C_{CT}

1.1-a
```
<flightQuery>
   <departureDate>2009-5-18</departureDate>
   <returnDate>2009-5-24</ returnDate>
   <fromCity>HongKong</fromCity>
   <toCity>Vancouver</fromCity>
   <airlineName>UnitedAirline</airlineName>
<flightQuery>
```

1.1-b
```
<flightQuery>
    ……
    <airlineName>-</airlineName>
<flightQuery>
```

2.1
```
<flightQueryReply>
    ……
   <airlineNameList>
      <airlineName>UnitedAirline</airlineName>
      <airlineName>AirCanada</airlineName>
   </airlineNameList>
<flightQueryReply>
```

XML Messages for C_{TF}

1.2-a
```
<hotelQuery>
   <departureDate>2009-5-18</departureDate>
   <returnDate>2009-5-24</ returnDate>
   <City>Vancouver</toCity>
   <hotelName>ShangriLa</hotelName>
<hotelQuery>
```

1.2-b
```
<hotelQuery>
    ……
   <hotelName>-</hotelName>
<hotelQuery>
```

2.2
```
<hotelQueryReply>
    ……
   <hotelNameList>
      <hotelName>Westin</hotelName>
      <hotelName>ShangriLa</hotelName>
   </hotelNameList>
<hotelQueryReply>
```

XML Messages for C_{TH}

(c) Typical XML messages exchanged between services

Figure 3. Choreography example of *TripHandling*.

a rectangle. A node in a rectangle represents a workflow activity of the corresponding service, and a link represents a transition between two activities. We further use a dashed line to represent a message that communicates among different roles of services in a choreography application.

The messages between two services are defined by different XML schemas. For instance, the XML messages tripQuery$^{1\text{-a}}$ and tripQuery$^{1\text{-b}}$ in Figure 3(c) are both defined by the schema tripQuery^{1}, and the XML message hotelQuery$^{2.2}$ in the same figure is defined by the schema hotelQuery$^{2.2}$. To save space, we use "…" to represent obvious or irrelevant lines of code. For example, the omitted lines in tripQuery$^{1\text{-b}}$ are the same as the *departureDate*, *returnDate*, *fromCity*, and *toCity* lines in tripQuery$^{1\text{-a}}$.

Given an XPath query q and an XML message m, let w denote the XML schema (in a WSDL specification) that defines m. In such situations, $q(m)$ may retrieve from m a dataset whose tags are defined in w. We further depict this set of tags as a circle or ellipse in Figure 4.

Case 1: In general, two XML schemas may contain tags having the same name. Suppose X_1 and X_2 represent the sets of tags contained in two XML schemas. We have three cases, namely X_1 and X_2 being partially overlapping, disjoint, and one is a subset of another. These three scenarios are depicted, respectively, as the *intersection*, *exclusion*, and *inclusion* relations in Figure 4. We also use schemas from the motivating example to illustrate each relation in the figure.

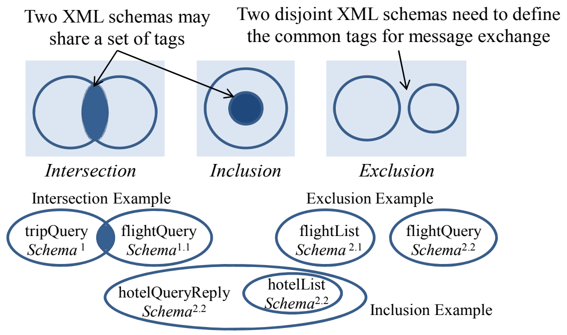


Figure 4. Relations among XML schemas in choreography.

However, two tags having the same name may come from different nodes of the same XML schema (or different XML schemas). Furthermore, a node may be reachable by multiple XPaths. A service may assume that it sends the content under a tag retrievable by one XPath, but its collaborating service may assume that it collects the content under a tag retrievable by another XPath. If these two XPaths do not refer to the same content of the XML message when applying their respective XML schemas, there will be an integration failure (or a mismatch between the two services in their message collaboration) even through the XML message may have successfully reached the second service.

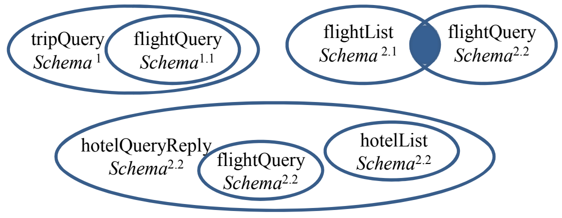
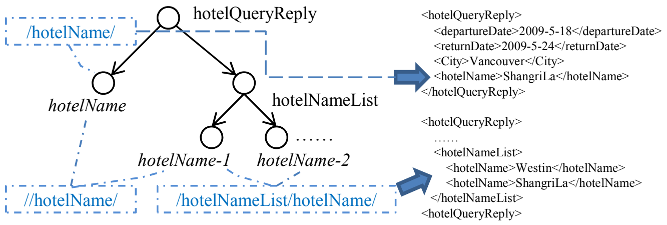


Figure 5. Faulty relations among XML schemas in choreography.

For example, each of the schemas in Figure 4 may mistakenly overlap with other schemas used in the same collaboration step or different steps (as shown in Figure 5 and annotated by the superscripts in the schema names). For instance, step 2.2 of the motivating example may mistakenly include the *flightQuery* schema inside the *hotelQueryReply* schema. Thus, data that should not be referred to by peer services will now be accessible, which may result in collaboration failures. To detect this problem, a technique should not only determine the inclusion, exclusion, and intersection relations, but also distinguish the amounts of overlapping. Say, if there are five overlapping tags, a good technique should test each of them in turn.

Nevertheless, although we have developed XRGs, a data structure to reveal the structure of XPath in the presence of schema, it is still generally infeasible to solve this problem. This is because covering all paths in an XRG merely means that some (but not necessary all) tags in individual XRG nodes have been exercised by a test case. The problem is further explained by the other two cases that follow. Thus, whether the content in relation to a particular tag can be successfully transferred from one service to another is not enforced by a technique at the XRG level.

Case 2: Take *tripQueryReply* in Figure 3 as an example. When *tripQuery* uses a concrete value of the "hotelName" tag (such as Westin) as a parameter, its reply will contain either exactly one hotel name or no name. However, when *tripQuery* does not supply any "hotelName" (by using "–" to query all possible hotel names), the reply will be a name list, where each listed item contains a "hotelName" tag and its value. Therefore, the same XPath query (such as `//hotelName/`) on these two types of XML messages can retrieve elements in different locations of an XML schema (as illustrated in Figure 6). On the other hand, `/hotelName/` and `/hotelQueryReply/hotelName/` will access different nodes, and yet they share the same tag name (also illustrated in Figure 6).



```
<hotelQueryReply>
  <departureDate>2009-5-18</departureDate>
  <returnDate>2009-5-24</returnDate>
  <City>Vancouver</City>
  <hotelName>ShangriLa</hotelName>
</hotelQueryReply>

<hotelQueryReply>
  ……
  <hotelNameList>
    <hotelName>Westin</hotelName>
    <hotelName>ShangriLa</hotelName>
  </hotelNameList>
<hotelQueryReply>
```

Figure 6. Effect of different data structures of XML messages in XPath query.

Case 3: Moreover, two different nodes with different tag names may refer to the same content. For example, the "toCity" tag in the "tripQuery" schema should refer to the "city" tag in the "hotelQuery" schema. These tags have been highlighted in Figure 3(b). The XPath queries `//city/`, `/tripQuery/fromCity/`, and `/tripQuery/toCity/` will return a city name in string format. If they are used in different choreography steps, these steps will implicitly be strongly coupled. Using techniques such as type checking on the schemas may not effectively distinguish such differences in the corresponding query results. However, the use of the former two XPath queries may extract the departure city rather than the destination city. In case there are hotels in the same chain in both the departure city and destination city (such as Westin Hotel), the hotel service may book a room in the hotel of the same name in the departure city rather than in the destination city, which will result in a failure.

In summary, the testing challenge illustrated by this example is that the sequence of XPath queries in the same step or different steps may raise different expectations on how to manipulate an XML message, which leads to failures in choreography collaboration.

4. OUR CHOREOGRAPHY MODEL

This section presents our formal model to facilitate data flow testing of choreography applications.

4.1 The C-LTS Model

As introduced in Section 2.2, LTS is a formal model to represent message interactions for scenario-based specifications. Therefore, we use LTS to model the skeleton of a choreography application. In this section, we enhance it gradually to realize our model of choreography applications. In LTS, when performing an action via message passing from a sender service to a receiver service, the transition that represents the action (which is a service invocation in choreography) is inadequate to represent the roles of XPath and WSDL in manipulating the required contents of the XML message. We propose, therefore, to extend LTS by attaching an XRG to every transition associated with an XPath and its document model.

We also assume that every transition represents either a send operation or a receive operation but not both.

Definition 4 (Labeled Query). A *labeled query* q for a choreography application P is an ordered couple $\langle \delta, r \rangle$ such that δ is a transition representing the action taken by P, and r is an XRG that models the sending and receipt of XML data by P.

For ease of presentation, we associate every transition δ with a labeled query. Even if δ is not originally associated with any XPath, we can attach an XPath that selects the entire contents of δ.

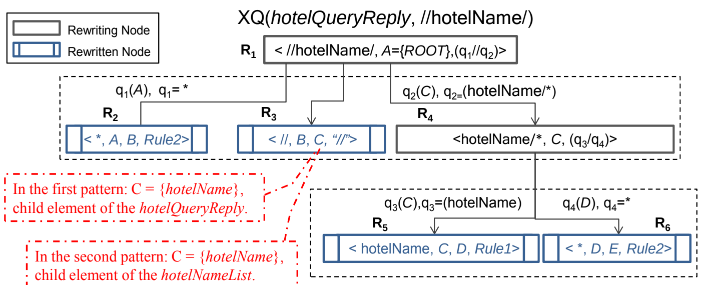
By modeling each XPath query and its document model as an XRG, we can use data flow analyses on the conceptual variables of the XRG (see Section 2.3 or [14]) to analyze choreography applications. Moreover, our model can also facilitate (data flow) testing at the inter-XPath or inter-WSDL levels. To do so, we propose to analyze service chorography with respect to whether the sets of tags at an XRG node are partially overlapping, completely overlapping, or disjoint from those at a node in another XRG.

A conceptual variable in an XRG (such as L^c and L^n in the definition of rewritten nodes in Section 2.2) is also a set of tags of an XML schema. Thus, in general, such a conceptual variable may contain multiple tags of an XML schema. If this is the case, as long as the corresponding XML message matches at least one tag, the variable does not distinguish the selected tag from others. Therefore, as explained in Section 3, simply adopting XRGs from [14] still cannot address the challenging issues illustrated in the motivating example. We thus propose a notion of XRG patterns as follows. For ease of presentation, we will use $r.A$ to denote the attribute A of r.

Definition 5 (XRG Pattern). For any given XRG $r = \langle q, \Omega, N_x, E_x, V_x \rangle$, an *XRG pattern* $\xi(r)$ is an instantiation of r such that (i) a tag t_i is assigned to the i-th variable ($\in V_x$) in a conceptual path (with all rewritten nodes in N_x included) based on the definition order of the variables, and (ii) t_i must be used (in the sense of data flow associations) by a subsequent rewritten node $n \in r.N_x$ to locate t_{i+1} in the conceptual path. The set of all XRG patterns of r is denoted by $\xi(\Re)$.

Intuitively, for any XRG r and any XRG pattern $\xi(r)$, L^c of each rewritten node n ($\in r.N_x$) in $\xi(r)$ contains exactly one tag. This is quite different from L^c of a rewritten node in XRG or its XRG patterns, which may contain multiple tag values. Moreover, by defining $\xi(\Re)$ as a set, it helps model the fact that every XRG pattern $\xi(r)$ with respect to $\Re$ is distinct.

We provide an example in Figure 7 to illustrate an XRG pattern. The figure presents an XRG for the XPath query "//hotelName/". The corresponding XML schemas are defined by the WSDL documents in Figure 3(b). In this example, we define five variables a, b, c, d, and e, representing the possible tags in the sets A, B, C, D, and E. With the exception of C, every set contains only one tag. We further identify two XRG patterns for C, as highlighted by the dashed text boxes. Since the tags for each conceptual variable can be statically computed during the construction of XRG, we can generate all possible candidate XRG patterns, and then eliminate the non-legitimate candidates according to Definition 5. Such a procedure for identifying XRG patterns can be done automatically.



Rewriting Node
Rewritten Node
XQ(hotelQueryReply, //hotelName/)
R1
< //hotelName/, A={ROOT},(q1//q2)>
q1(A), q1= *
q2(C), q2=(hotelName/*)
R2
< *, A, B, Rule2>
R3
< //, B, C, "//">
R4
<hotelName/*, C, (q3/q4)>
In the first pattern: C = {hotelName}, child element of the hotelQueryReply.
In the second pattern: C = {hotelName}, child element of the hotelNameList.
q3(C),q3=(hotelName)
q4(D), q4=*
R5
< hotelName, C, D, Rule1>
R6
< *, D, E, Rule2>

A = {*ROOT*} *B* = {*hotelQuery*} *C* = {*hotelName*, *hotelNameList*, City, …} *D* = {*HotelName*}

E = {*e* | *e* is the *hotelName* value}

Rule 1: $n(x) = \{y | (x, y) \in EDGES(t), LABEL(y) = n\}$ *Rule 2*: $*(x) = \{y | (x, y) \in EDGES(t)\}$

Rule 4: $(q1/q2)(x) = \{z | y \in q1(x), z \in q2(y)\}$ *Rule 5*: $(q1//q2)(x) = \{z | y \in q1(x), (y, u) \in EDGES^*(t), z \in q2(u)\}$

"//": $//(x) = \{y | (x, y) \in EDGES^*(t)\}$

Figure 7. Example of two XRG patterns.

Individual XRG patterns represent valid message types for data manipulation. A pair of send and receive operations at the service endpoints are thus annotated with XRG patterns so that we can resolve how messages can be routed through a pair of XML definitions. Based on the notion of XRG patterns, we present a formal model that allows us to further the study of data flow modeling, testing, and analysis of choreography applications.

Definition 6 (LTS-Based Choreography (C-LTS) Model). A *C-LTS model* for a choreography application P is a 6-tuple $\langle S, L, \Delta, s, V, \Re \rangle$ satisfying the following:

- $\langle S, L, \Delta, s \rangle$ is an LTS of P.
- V is a set of variables such that every $v \in V$ is a variable defined in the choreography program P to serve as the input parameter or return value for a labeled query q, or is a conceptual variable defined in an XRG r ($\in \Re$), that is, $v \in r.V_x$.
- $\Re$ is a set of XRGs in P such that every transition $\delta \in \Delta$ is associated with a labeled query $q = \langle \delta, r \rangle$ if and only if $r \in \Re$.

An execution for a C-LTS model is a sequence $e = \langle s_1, q_1, s_2, q_2, \ldots \rangle$ such that (a) $s_1 = s$, (b) $\langle s_{i-1}, q_i.l, s_i \rangle \in \Delta$ for $i = 2, \ldots, |e|$, and (c) every q_i is a couple $\langle l, cp \rangle$, where $\delta = \langle s_i, q_i.l, s_{i+1} \rangle$ is a transition, $\langle \delta, r \rangle$ is a labeled query, and cp is a conceptual path of $\xi(r)$. To be consistent with the convention of an LTS execution, if δ is a send operation, then the fragment $\langle s_i, q_i.l, s_{i+1} \rangle$ can be rewritten as $\langle s_{i-1}, q_i.l, q_i.cp, s_i \rangle$. Similarly, if δ is a receive operation, then the fragment $\langle s_{i-1}, q_i.l, s_i \rangle$ can be rewritten as $\langle s_{i-1}, q_i.cp, q_i.l, s_i \rangle$. The corresponding test trace for the C-LTS is a projection of the execution on the alphabet of P (see Definition 2).

4.2 Data Flow Entities for Choreography

This section proposes a family of data flow testing criteria to measure the quality of test sets for choreography applications.

4.1.1 Conventional Data flow Associations

In this section, we revisit the basic definitions in data flow testing [8] to make the paper self-contained. A *Control Flow Graph* (*CFG*) for a program unit is an ordered couple $\langle V, E\rangle$. V is a set of nodes representing statements. E is a set of directed edges representing transitions among statements. A *complete path* in a CFG is a path starting from the entry node and ending with an exit node. A *sub-path* is part of a complete path. A *definition* of variable x at node n occurs when either the value of x is stored or updated at n. A *usage* of variable x at node n occurs when either the value of x is fetched or referenced at n.

A *definition-clear* path with respect to a variable x is a path in which none of the nodes (except the first and the last nodes) defines or undefines x. A *def-use* association is a triple $\langle x, n_d, n_u\rangle$ such that the variable x is defined at node n_d and used at node n_u, and the path from n_d to n_u is definition clear with respect to x. For ease of presentation, we follow [11] and define a predicate *def_clear*($\langle x, n_d, n_u\rangle$) to be true if and only if $\langle x, n_d, n_u\rangle$ is a def-use association.

4.1.2 Data Flow Associations for Choreography

This section discusses def-use associations in our C-LTS model. A C-LTS model may contain two types of variables: choreography variables that have been defined in choreography programs (see [25], for example), and conceptual variables that have been defined in XRGs [14]. We define the *definition* and *use* of these variables in our choreography model in Definition 7.

Definition 7 (*Chore-Query-Def* and *Chore-Query-Use* of Variables). Given a C-LTS model $P = \langle S, L, \Delta, s, V, \mathfrak{R}\rangle$,

- A *chore-query-def* (*def*$_{cq}$) of a variable $v \in V$ is either (i) an occurrence of v in an action associated with a transition $\delta \in \Delta$ such that v is *assigned* the return value (i.e., the extracted content from an XML message) of an XPath query, or (ii) a definition occurrence of v at node n of an XRG r ($\in \mathfrak{R}$).
- A *chore-query-use* (*use*$_{cq}$) of a variable $v \in V$ is either (i) an occurrence of v in an action associated with a transition $\delta \in \Delta$ such that v is *used* as an input parameter of an XPath query associated with δ, or (ii) a use occurrence of v at node n of an XRG r ($\in \mathfrak{R}$).

We present further definitions to cover different XML-manipulating data structures.

Definition 8 (*Chore-Query-Pattern-Def* and *Chore-Query-Pattern-Use* of Variables). Given a C-LTS model $P=\langle S, L, \Delta, s, V, \mathfrak{R}\rangle$,

- A *chore-query-pattern-def* (*def*$_{cpq}$) of a variable $v \in V$ is either (i) a triple $\langle v, \delta, \xi(r)\rangle$ for some $r \in \mathfrak{R}$ such that there is an occurrence of v in the action associated with a transition $\delta \in \Delta$ and v is *assigned* the return value of $\xi(r)$, or (ii) a triple $\langle v, \delta, \xi(r)\rangle$ for some $r \in \mathfrak{R}$ such that there is a definition occurrence of v in a rewritten node n of $\xi(r)$.
- A *chore-query-pattern-use* (*use*$_{cpq}$) of a variable $v \in V$ is either (i) a triple $\langle v, \delta, \xi(r)\rangle$ for some $r \in \mathfrak{R}$ such that there is an occurrence of v in an action associated with transition $\delta \in \Delta$ and v is *used* as an input parameter of $\xi(r)$, or (ii) a triple $\langle v, \delta, \xi(r)\rangle$ for some $r \in \mathfrak{R}$ such that there is a usage occurrence of v in a rewritten node n of $\xi(r)$.

Next, we demonstrate how we formulate def-use associations on top of our C-LTS model. We define two kinds of def-use associations in our model: Given a C-LTS model $P = \langle S, L, \Delta, s, V, \mathfrak{R}\rangle$, a *chore-query-def-use* association α for a variable v is a triple $\langle v, n_d, n_u\rangle$ such that v is a *def*$_{cq}$ at n_d and a *use*$_{cq}$ at n_u, and there is a definition-clear sub-path with respect to v from n_d to n_u. Similarly, we define a *chore-query-pattern-def-use* association as a triple $\langle v, n_d, n_u\rangle$ such that v is a *def*$_{cpq}$ at n_d and a *use*$_{cpq}$ at n_u, and there is a definition-clear sub-path with respect to v from n_d to n_u.

The algorithm to construct these data flow entities can be straightforwardly developed based on the definitions of these entities. Owing to space limit, we omit the algorithm in this paper.

4.3 Test Adequacy Criteria for Choreography

This section proposes a family of data flow testing criteria to measure the quality of test sets to test choreography applications. Our first test criterion is to exercise every XRG on each LTS test trace at least once. Such an adequate test set should cover all XRGs on all LTS test traces in the choreography application under test.

Criterion 1 (*All Chore-Queries*). A test set T satisfies the all-*chore-queries* criterion for a C-LTS model $P = \langle S, L, \Delta, s, V, \mathfrak{R}\rangle$ if and only if every XRG r ($\in \mathfrak{R}$) is exercised by at least one test case $t \in T$.

Executing a *chore-query* once may not help evaluate all *chore-query-def-use* associations for conceptual variables in an XRG. Therefore, we continue to explore the structure of the XRG. It requires a test set to cover all *chore-query-def-use* associations.

Criterion 2 (*All Chore-Query-Uses*). A test set T satisfies the all-queries criterion for a C-LTS model $P = \langle S, L, \Delta, s, V, \mathfrak{R}\rangle$ if and only if, for each *chore-query-def-use* association α, there is at least one test case $t \in T$ such that *def_clear*(α) is evaluated to be true.

Finally, we define Criterion 3 to cover all *chore-query-pattern-def-use* associations. We note that *all-chore-query-uses* subsumes [8] *all-chore-queries* because every XPath query is evaluated by at least one *chore-query-use*. Similarly, any *chore-query-use* is evaluated by at least one *chore-query-pattern-use*, and hence *all-query-pattern-uses* subsumes *all-chore-query-uses*.

Criterion 3 (*All Chore-Query-Pattern-Uses*). A test set T satisfies the *all-chore-query-pattern-uses* criterion for a C-LTS model $P = \langle S, L, \Delta, s, V, \mathfrak{R}\rangle$ if and only if, for each *chore-query-pattern-def-use* association α, there is at least one test case $t \in T$ such that *def_clear*(α) is evaluated to be true.

Definition 8 and Criterion 3 recognize that, during testing, we need to consider how the contents of a tag may transfer from one XRG to another XRG.

5. EVALUATION

In this section, we evaluate our approach through a case study.

5.1 Experiment Setup

Our case study uses the *Data Exchange Platform* (*DEP*) application, which is a part of the *University Resource Planning* (*URP*) project implemented in a university outside Hong Kong. URP can be viewed as a downsized version of an Enterprise Resource Planning (ERP) application (such as SAP). We choose DEP because we are allowed to access the source code for academic experimentation. The original version of DEP is a Java application.

We adapt DEP using WS-CDL (presently the most popular service choreography language) [22] to replace the existing Java-based interface for communicating among web services. More

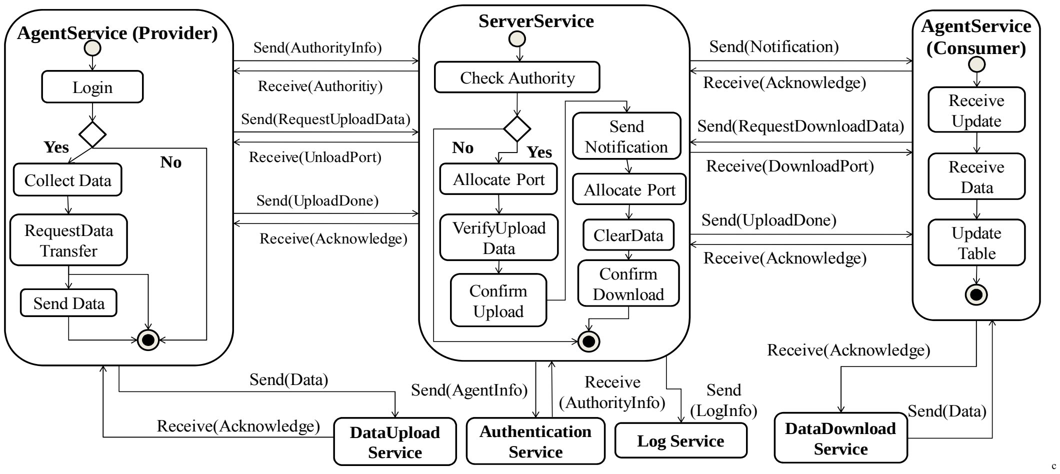


Figure 8. UML activity diagram showing typical scenarios for Data Exchange Platform.

specifically, for each function (see the next paragraph), we mechanically translate the functional signature into a portType, and translate the parameter types into an XML schema. If a parameter is an object, the XML schema will contain an object ID field to identify the object. XPaths are also mechanically specified so that such a portType can select the objects and values from the XML schema as if the original version accepted Java objects and values. We further examine the source code of the original version of DEP to identify all the call sites of such a function, and specify the port relations between two services according to every identified call site in a WS-CDL specification.

Owing to our limited resources, we select the four biggest services (in terms of lines of Java code in the original version) for the verification of our proposal in the case study. The four subject services are summarized below and illustrated as a UML activity diagram in Figure 8.

- *AgentService*. Multiple agent services are distributed in different information systems. This service monitors the database updates, collects the change logs, and collaborates with *MonitorService* to update the data stored in other information systems according to the change log.
- *MonitorService* handles the requests from *AgentService*, verifies the authority of the agent (by communicating with *Autheticationservice*), and allocates a data transfer thread to handle the authenticated request.
- *DataService* consists of two subservices: *DataUploadService* enables an agent to upload data to the server, and *DataDownloadService* enables an agent to download data from the server.
- *AutheticationService* authenticates whether an agent has the rights to perform the data transfer.

The statistics of the subject services are summarized in Table 1.

Table 1. Descriptive statistics of the subject services.

Services	No. of Ports	WSDL	XPath	LOC (Java)	No. of Faults (in XPath and WSDL of Services)
AgentService	6	2	6	4,000–5,000	3
MonitorService	8	2	8	6,000–7,000	3
DataService	4	1	4	3,000–4,000	2
AutheticationService	2	1	2	1,000–2,000	2
Total	20	6	20	> 14,000	10

We have implemented a tool to compute the XRG patterns, and the proposed def-use associations for the case study. We generate different faulty versions by seeding one fault in each copy of the original program. We randomly select 10 faults from [14] and, for each of the selected faults, we randomly select one feasible position in the WSDL, XPath, or WS-CDL specification of DEP to simulate the fault. We create 10 faulty versions in total. We have also implemented a tool to randomly generate 1000 test cases to form a test pool for the case study based on the original adapted SOA version. Our tool then generates test suites for each of our testing criteria and for random testing. When generating each test suite for our testing criteria, the tool randomly selects a test case from a test pool and evaluates it on the application. We add this test case to the test suite only if the former can help improve the coverage specified by the criterion. This procedure will terminate if either 100% coverage of a criterion has been attained, or an upper bound of 1000 trials has been reached. We repeat the procedure 100 times for every version. We originally planned to use more faulty versions in the case study. However, the experiment is intricate to conduct, and owing to effort and resource limitations, we settle for the size of the reported experiment, and have not included other testing criteria in the case study. For random testing, we first randomly select a test suite whose size is the same as the largest test suite for our testing criteria on the same program version. We then construct another random test suite by whose size is the same as the smallest test suite for our testing criterion.

We observe from the procedure for test case selection for each criterion that random testing with the largest test suite (*max-size* for short) has the same size as the *all-chore-query-pattern-uses* criterion, and random testing with the smallest test suite (*min-size* for short) has the same size as the *all-chore-queries* criterion.

We choose the fault-detection rate [8] as the effectiveness measure in the experimentation, defined as the proportion of the number of test cases that reveal failures to the total number of test cases.

Table 2. Fault-detection rates of testing criteria

Criterion	Fault-Detection Rates		
	Min.	Avg.	Max.
Random (*Min-Size*)	0.200	0.512	0.900
Random (*Max-Size*)	0.400	0.667	1.000
All-Chore-Queries	0.400	0.684	0.900
All-Chore-Query-Uses	0.600	0.763	1.000
All-Chore-Query-Pattern-Uses	0.800	0.901	1.000

5.2 Data Analysis

We analyze the results of the case study in this section. Table 2 summarizes the minimum, average, and maximum fault-detection rates of each criterion. It shows that the *all-chore-query-pattern-uses* criterion is the most effective (90.1% on average) among all the criteria studied. For instance, it is better than random testing (*max-size*) by 23%. The average effectiveness of the *all-chore-query-uses* criterion is slightly (8%) better than the *all-chore-queries* criterion. The *all-chore-queries* criterion is 17% better than random testing (*min-size*) with the same number of test cases, and is similar in effectiveness to random testing (*max-size*). The result indicates that the XRG pattern we have proposed in the paper can be promising in improving the effectiveness of testing.

To further verify the effectiveness of our techniques, we order the faulty versions in ascending order of their fault-detection results reported by random testing (*max-size*) obtained from the experiment above. We then construct 10 faulty program suites as follows: Suite #1 contains only one program, which is the faulty version assigned the lowest fault-detection rate by random testing. Suite #2 contains the two faulty versions that have been assigned the lowest two fault-detection rates. In general, Suite #*n* (*n* = 1, 2, ..., 10) contains the *n* faulty versions that have been assigned the lowest *n* fault-detection rates by random testing. For each testing criterion and for each of the faulty program suites, we then compute the mean fault-detection rate for all programs in the suite. The results are shown in Figure 9.

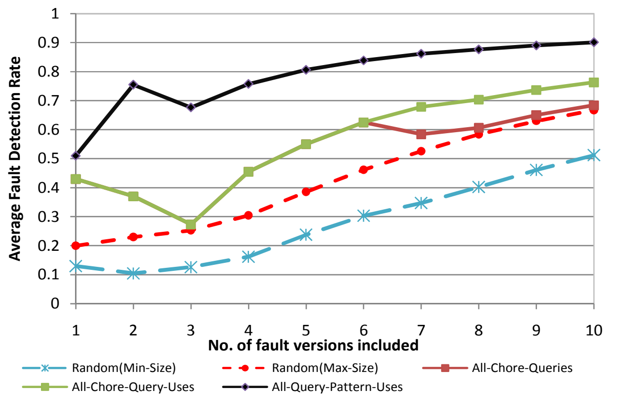


Figure 9. Comparisons of different faulty program suites.

The figure shows that the *all-chore-queries* criterion is also better than random testing (*max-size*) in revealing faults that are more difficult to be detected (such as when *n* = 1 to 6). For most versions, we observe that *all-chore-query-uses* is more than 10 percent better than random testing (*max-size*) in terms of the fault-detection rate. Our *all-query-pattern-uses* criterion is significantly better than random testing in all experimented cases. In particular, for the 2nd faulty program suite, the effectiveness of *all-chore-query-pattern-uses* is above 75%, whereas that of random testing (*max-size*) is lower than 25%. Furthermore, we observe that *all-chore-query-pattern-uses* is more effective (around 40%) than *all-chore-query-uses* in detecting faults in the 2nd and 3rd faulty program suites.

In summary, the experimental results show that our technique using XRG patterns can detect more than 90% of all faults, which is encouraging. In the future, we will study how to detect subtle faults more effectively, and conduct multi-fault experiments and compare our technique with other testing criteria.

5.3 Threats to Validity

This section discusses the threats to validity of the experiment. Threats to internal validity are the influences that can affect the dependency of the experimental variables involved. When executing a test case, the contexts of the involved services (e.g., database conditions) may affect the result, making the result nondeterministic. The problems of service composition raised by contextual environments have been discussed in [13]. To address this problem, our experiment tool confirms whether the contexts of services are reproducible for every test case (by resetting the contexts to the same values each time and rerunning the test case). Moreover, we simulate a multi-fault version in the experiment. It is less desirable than using real multi-fault versions to conduct the experiment, and using the latter may give other results. However, given that our technique outperforms random testing to a large extent, we believe that our techniques can be effective on real-life versions.

External validity refers to whether the experiment can be generalized. We use a case study to evaluate our approach. We choose this application because we can access the source code to adapt and study testing techniques, and because we are not aware of representative open-source service-oriented applications available for evaluation. However, the scale of the case study is not large, even though we have spent much effort on the experiment. In the future, we plan to use other subject programs to study the fault-detection effectiveness of the testing criteria to gain more insight. Moreover, we follow WS-CDL specifications to use WSDL to define XML messages and to use XPath as expressions to extract required contents from these XML messages. It may not be representative if XPath, WSDL or WS-CDL is not used in the subject applications.

6. DISCUSSIONS

We model a choreography application on top of Labeled Transition Systems (LTS), and develop a C-LTS model to facilitate data flow testing for choreography applications. However, our methodologies of using XRG and XRG patterns for modeling conceptual content selection in an XML document are not limited to LTS. Other models that use XPath (on choreography) may also apply our methodologies.

Second, when defining our C-LTS model, we consider each participated service in the choreography as a black-box component, and have not considered the program structures of individual services. In general, a service participating in the choreography can be an orchestration service or choreography service. (i) For an orchestration service, one may extend or model by incorporating the control flow graph of the orchestration service, and then incorporate the data flow entities (similar to [14]) in addition to the data flow entities based on C-LTS. (ii) For a choreography service, one may model it using our C-LTS model, and extend test traces similarly to how process algebras (such as CSP) serialize concurrent processes into trace sets. Moreover, a service may further participate in multiple choreography applications. We plan to study how to include choreography information in the modeling and testing of orchestration applications [13].

Third, our C-LTS model has taken XPath and WSDL into account. WSDL is defined by the World Wide Web Consortium (W3C) and has been widely adopted as the standard to define web services in service-oriented applications. WS-CDL [22], defined by W3C is also a popular specification in designing service collaborations among applications. As stated in the WS-CDL document, XPath must be used in specifying expressions, queries, and predicates in WS-CDL. Therefore, our model can be applied to WS-CDL applications in general.

7. RELATED WORK

In this section, we review the work related to ours. We first briefly review the research on modeling service composition. Brambilla et al. [2] proposed a process model aiming to achieve an effective high-level specification of web applications featuring business processes and remote services invocations. Roman et al. [18] proposed a semantic model for web service choreography. Our techniques aim at testing the functional correctness of service choreography and do not require a full semantic model of services.

Next, we review the research efforts on analyzing the properties of service composition. Foster et al. [6] found that model checking approaches which ignore resource constraints of the deployment environment are insufficient to establish safety and liveness properties of services (such as the identification of deadlocks caused by complex interplays between services and execution hosts.) They proposed to link services and resource management to solve such problems. Ye et al. [27] studied the atomicity of service composition. By using the encapsulated details, they analyzed the implicit interactions among services in service composition. Their approach aims to detect the atomicity violations of service composition.

Many researchers have proposed techniques to test service-oriented applications. Chan et al. [4] applied metamorphic relations to construct test cases for stateless web services. Li et al. [10] studied unit testing problems for service orchestration. Our previous work [14] studied the complexity raised by XPath and WSDL in integrating different flow steps in service orchestration. In this paper, we study the testing problem from the choreography perspective rather than from the orchestration perspective. Fu et al. [9] translated web services into Promela for formal verification. They translated an XPath into a Promela procedural routine using self-proposed variables and code to simulate XPath operations. We translate an XPath strictly following the definition of XPath expressions given in [16] into an XPath Rewriting Graph. On top of [14], we propose XRG patterns to enrich the concept of XRGs.

Finally, we briefly review data flow testing techniques. Many existing techniques [8][11] on data flow testing are based on information obtained from program code without considering artifacts like XPath and WSDL. Bartolini et al. [1] discussed potential ways to apply data flow testing to service compositions in a general sense. Mei et al. [14] modeled XPath and WSDL in the XRG, and developed data flow testing techniques for orchestration applications. However, how XRG can be adapted to facilitate data flow testing on top of a choreography model has not been addressed.

8. CONCLUSION

In SOA, choreography is a strategy that specifies how services collaborate. The messages from one service to another may, however, require the use of many XPaths to manipulate or extract the message contents. Mismatches in XPath manipulation between the sender service and the choreography specification, within the choreography specification to relate incoming and outgoing messages, or between the choreography specification and the receiver service may result in failures in service choreography.

In this paper, we have proposed a C-LTS model to represent choreography applications from the testing perspective. To model message exchanges between services in a choreography application, we have proposed to annotate each action available in the WS-CDL and WSDL interface with the associated XPath queries and WSDL specifications (in the format of XPath Rewriting Graphs (XRGs)). Moreover, to address the challenges such as mismatches in message content selection, we have proposed the notion of XRG patterns to explicitly model how message contents can be unified despite the multiple interpretations by different XML schemas. Since the actual unification is conducted through concrete tag matching available in the choreography application, we have developed an algorithm to automatically generate all the required XRG patterns. We have further used the XRG patterns as variables to analyze how the message contents of tags associated with an XML schema may be used by other XRGs or XRG patterns, and have identified new data flow associations. We have thus proposed a new family of data flow testing criteria for choreography applications, and evaluation them in an empirical case study.

There is, of course, room for improvement. For example, we will study how to incorporate orchestration information into our model. We will also study how to integrate the notion of contexts into the testing of service-oriented applications.

9. REFERENCES

[1] C. Bartolini, A. Bertolino, E. Marchetti, and I. Parissis. Data flow-based validation of web services compositions: perspectives and examples. In *Architecting Dependable Systems V*, pages 298–325, 2008.

[2] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 15 (4): 360–409, 2006.

[3] M. Broy, I. H. Krüger, and M. Meisinger. A formal model of services. *ACM TOSEM*, 16 (1): Article No. 5, 2007.

[4] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *Proceedings of QSIC 2005*, pages 470–476, 2005.

[5] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM TOSEM*, 10 (1): 56–109, 2001.

[6] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In *Proceedings of ESEC/FSE 2007*, pages 225–234, 2007.

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings ASE 2003*, pages 152–161, 2003.

[8] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14 (10): 1483–1498, 1988.

[9] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of WWW 2004*, pages 621–630, 2004.

[10] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. BPEL4WS unit testing: framework and implementation. In *Proceedings of ICWS 2005*, pages 103–110, 2005.

[11] H. Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *Proceedings of SIGSOFT 2006/FSE-14*, pages 242–252, 2006.

[12] L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and regression testing of COTS-component-based software. In *Proceedings of ICSE 2007*, pages 85–95, 2007.

[13] L. Mei. A context-aware orchestrating and choreographic test framework for service-oriented applications. In *Proceedings of ICSE Companion 2009* (*Doctoral Symposium*), pages 371–374, 2009.

[14] L. Mei, W. K. Chan, and T. H. Tse. Data flow testing of service-oriented workflow applications. In *Proceedings of ICSE 2008*, pages

371–380, 2008.

[15] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of WWW 2009*, pages 901–910, 2009.

[16] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51 (1): 2–45, 2004.

[17] C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36 (10): 46–52, 2003.

[18] D. Roman and M. Kifer. Semantic web service choreography: contracting and enactment. In *The Semantic Web — ISWC 2008*, pages 550–566, 2008.

[19] *State Transition System*. Wikipedia. Available at http://en.wikipedia.org/wiki/State_transition_system. (Last access on June 16, 2009.)

[20] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *Proceedings of ESEC 2001 / FSE-9*, pages 74–82, 2001.

[21] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *Proceedings of SIGSOFT 2002 / FSE-10*, pages 109–118, 2002.

[22] *Web Services Choreography Description Language Version 1.0*. W3C, 2005. Available at http://www.w3.org/TR/ws-cdl-10/.

[23] *Web Services Choreography in Practice*. IBM, 2005. Available at http://www.ibm.com/developerworks/xml/library/ws-choreography/.

[24] *Web Services Description Language* (*WSDL*) *2.0*. W3C, 2007. Available at http://www.w3.org/TR/wsdl20.

[25] *WS-CDL Eclipse* (*with Examples*). Sourceforce.net, 2005. Available at http://sourceforge.net/project/showfiles.php?group_id=138675.

[26] *XML Path Language* (*XPath*) *Recommendation*. W3C, 2007. Available at http://www.w3.org/TR/xpath20/.

[27] C. Ye, S. C. Cheung, W. K. Chan, and C. Xu. Detection and resolution of atomicity violation in service composition. In *Proceedings of ESEC/FSE 2007*, pages 235–244, 2007.