Data Flow Testing of Service-Oriented Workflow Applications[*]

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

WS-BPEL applications are a kind of service-oriented application. They use XPath extensively to integrate loosely-coupled workflow steps. However, XPath may extract wrong data from the XML messages received, resulting in erroneous results in the integrated process. Surprisingly, although XPath plays a key role in workflow integration, inadequate researches have been conducted to address the important issues in software testing. This paper tackles the problem. It also demonstrates a novel transformation strategy to construct artifacts. We use the mathematical definitions of XPath constructs as rewriting rules, and propose a data structure called *XPath Rewriting Graph* (*XRG*), which not only models how an XPath is *conceptually rewritten* but also tracks individual rewritings progressively. We treat the mathematical variables in the applied rewriting rules as if they were program variables, and use them to analyze how information may be rewritten in an XPath conceptually. We thus develop an algorithm to construct XRGs and a novel family of data flow testing criteria to test WS-BPEL applications. Experiment results show that our testing approach is promising.

Categories and Subject Descriptors

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

General Terms: Measurement, Reliability, Verification

Keywords: WS-BPEL, XPath, service-orientation, workflow testing, testing, rewriting rules, SOA, XML, XML document model

1. INTRODUCTION

Software engineers often employ a collection of heterogeneous but closely related technologies, such as WS-BPEL [1], to develop a service-oriented workflow application [4]. They may design a company's business workflow in BPEL [1], or source (external) web services [2] to provide functions of individual workflow steps. Furthermore, they specify the signatures and resource locators (such as URLs) of their web services as WSDL documents [1] so that BPEL can invoke these web services. To facilitate XML-based communications and data transfers among web services and individual BPEL steps, software engineers may define the required message types by using XML schema [11]. Any concrete messages, WSDL documents, or definitions of BPEL variables are, however, XML documents. XPath [3] is an indispensable means to manipulate these documents, such as extracting the required contents from an XML message returned by a web service, or keeping the extracted contents in a BPEL variable under the right variable definition. For instance, every WS-BPEL application in an IBM Repository [4] uses XPath. A mismatch among components (e.g., extracting the wrong contents or failing to extract any content from a correct XML message) may cause a WS-BPEL application to function incorrectly.

Surprisingly, although WS-BPEL is strongly advocated by OASIS, IBM, Microsoft, BEA, SAP, and Oracle to be a platform for building enterprise applications, inadequate researches have been conducted to address their testing issues (see Section 6 for details). In particular, even though XPath plays such a crucial role in WS-BPEL applications, many existing verification and validation (V&V) researches (such as in modeling and verification [22][29], validation [25], unit testing [18], and test case generation [12][28]) simply do not consider XPath or merely model it as a function call without exploring deeply its implication from the V&V perspective. Some (e.g., [10][11]) simulate XPath expressions in another language using their styles of programming. The conceptual structure of XPath and how various parts of this structure may interact with BPEL remain unclear.

In a typical WS-BPEL application, for instance, XPath may work in pair with a document model of XML messages (that is, an XML schema [3]) to extract the required contents. Depending on the structure of the XML schema, however, multiple paths may fulfill the same XPath, but extract different contents from the same XML message. Nevertheless, even different entities specified in an XML schema may share the same primitive data types such as string, they may serve distinct purposes. Using incompatible (in the sense of semantics) extracted messages to conduct follow-up workflow activities in a BPEL program may result in integration errors. We shall give a motivating example in Section 2.1 to elaborate our point.

XPath should be studied deeply in testing research to improve the quality of WS-BPEL applications [16]. To our best knowledge, existing testing researches do not adequately address the interactions among XPath, XML schema, and XML messages, and their relationships with BPEL. In this paper, we study this problem and propose a solution to tackle the testing challenges.

As the use of XPath is fundamental in developing a WS-BPEL application, we firstly study how to reveal the implicit structure of XPath, which should be close to its declarative semantics so that it will not be biased to a particular BPEL engine implementation, as well as study the interactions between BPEL and XPath. Gottlob et al. [13] have shown that such paths are generally not decidable.

[*] This research is supported in part by grants of the Research Grants Council of Hong Kong (project nos. 111107, 123207, and 716507).

[†]All correspondence should be addressed to Prof. T.H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2858 4141. Email: thtse@cs.hku.hk.

They also propose a decidable fragment, and a set of definitions to capture the fragment.

Our model for XPath is built atop this fragment and these definitions of XPath syntactic constructs. We treat the definitions as "left-to-right" rewriting rules (similarly to the application of axioms as rewriting rules in algebraic specifications [6]). Through a series of application of these rules, we rewrite an XPath into a normal form, which means that no more rewriting rules can be applied, or a fixed point for recursive definitions has been reached.

Furthermore, instead of merely analyzing the (final) normal form, we record the series of (intermediate) rewriting results (see Section 3). As such, our model captures how each applicable rewriting rule uses its "left" part to unify with an XPath sub-expression (of an intermediate result) to construct the next level of intermediate results or the corresponding normal form via the "right" part of the rule.

We develop a data structure (dubbed *XPath Rewriting Graph* or *XRG* for short) to model an XPath in WS-BPEL. In the spirit of data flow testing and analysis [8][20][26], we further innovatively consider any variable generated as a *variable definition*, and the use of a variable provided by a preceding node as a *variable usage*. We note that such variables are conceptual in nature, and they are not program variables because they *never* appear in a program implementation. We thus term them as *conceptual variables*. Together with the inputs to an XPath from a BPEL program and its output variables defined to specify the data to be transferred back to BPEL, this data structure forms an explicit artifact to model different paths, conceptually defined in an XPath, on how to provide query values to BPEL programs.

By also modeling a BPEL program as a control flow graph, we propose an approach to identify the data flow associations relevant to the conceptual variables in XPath and ordinary variables in the BPEL program, and then formulate a set of test adequacy criteria to measure the quality of test sets.

The main contributions of this paper are multifold: (i) It demonstrates a novel strategy that transforms schema-based definitions, which are recursively defined, into explicit artifacts. (ii) A data structure, *XPath Rewriting Graph*, is proposed to model XPath at a conceptual level. (iii) This paper is among the first work on WS-BPEL testing that tackles the complexity of XPath. (iv) We identify a new type of dataflow entity to capture the characteristics of XPath. (v) We propose a family of test adequacy criteria to measure the quality of test sets. (vi) To our best knowledge, we provide the first set of experiments to evaluate the impact of XPath for services testing research using open-source programs. It shows that our approach is promising.

The rest of the paper is organized as follows: Section 2 outlines the technical preliminaries and testing challenges for WS-BPEL applications. Section 3 presents the algorithm of constructing an XRG, and our effort to model WS-BPEL applications. Section 4 introduces our data flow model and testing criteria to measure the comprehensiveness of test sets. Section 5 reports an experimental evaluation of our proposal, and followed by a literature review and conclusions in Sections 6 and 7, respectively.

2. WS-BPEL APPLICATIONS

This section presents a motivating example and introduces the technologies in typical WS-BPEL applications [1].

2.1 Motivating Example

Our motivating example to illustrate the challenges in the testing of WS-BPEL applications is adapted from the Apache WSIF project [27]. It involves a Digital Subscriber Line (DSL) application that offers DSL query services. Since the code (in XML format) is quite lengthy, we use an activity diagram in Figure 1 to depict the business process (*IsServiceAbailable*) of the example, in which each node denotes a BPEL activity, and each link denotes a transition between two activities. We also annotate the nodes with additional information, such as the input and output parameters of the activities, or any XPath Query used by the activities in the BPEL code. We number the nodes as A_1, A_2, …, A_8 to ease subsequent discussions. The service *IsServiceAbailable* is described as follows:

(1) A_1 invokes the service *AddressBookLookup*, which retrieves the address information from the address book by searching the given user name through the BPEL variable *UserName*, and stores the returned XML message in the BPEL variable *UserAddress*.

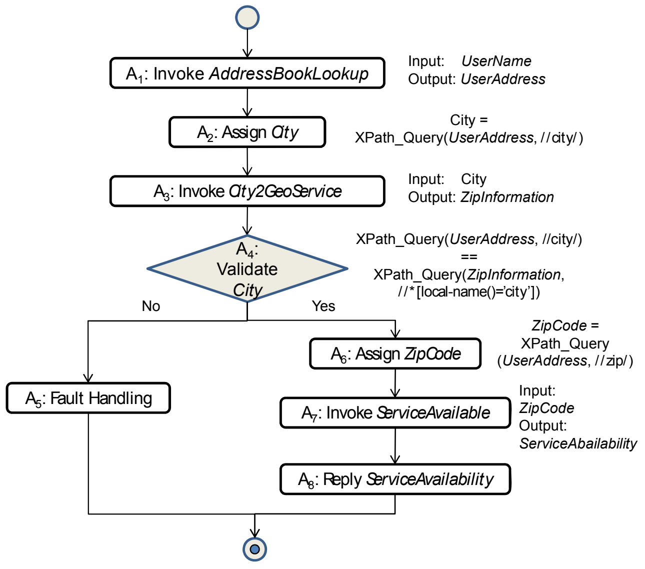
(2) A_2 extracts the city name from *UserAddress* via the XPath `//city/` and assigns the city name to the BPEL variable *City*.

(3) A_3 invokes the service *City2GeoService*, which looks up the zip information based on the given *City*, and keeps the result in the variable *ZipInforamtion*.

(4) A_4 checks whether the city name in *UserAddress* is the same as that in *ZipInformation* by extracting their city fields through the XPaths `//city/` and `//*[local-name()='city']`, respectively, where `local-name()` is an XPath function that returns the name of an element.

(5) If A_4 detects no problem, A_6 further extracts the zip code from *UserAddress* via the XPath `//zip/` and assigns it to the BPEL variable *ZipCode*. Then, A_7 executes the service *ServiceAvailable* to obtain the service availability status, and finally A_8 returns the *ServiceAvailability* information to the caller.

(6) If A_4 detects a problem, A_5 will execute a fault handler.



A1: Invoke AddressBookLookup
Input: UserName
Output: UserAddress
A2: Assign City
City = XPath_Query(UserAddress, //city/)
A3: Invoke City2GeoService
Input: City
Output: ZipInformation
A4: Validate City
XPath_Query(UserAddress, //city/) == XPath_Query(ZipInformation, //*[local-name()='city'])
No
Yes
A6: Assign ZipCode
ZipCode = XPath_Query (UserAddress, //zip/)
A5: Fault Handling
A7: Invoke ServiceAvailable
Input: ZipCode
Output: ServiceAbailability
A8: Reply ServiceAvailability

Figure 1. Business Process *IsServiceAvailable*

The definition of the structure of any BPEL variable is kept in an XML schema. For example, the variables *UserAddress* and *ZipInformation* in Figure 1 are defined by the schemas *address* and *LatLongReturn*, respectively, in Figure 2. The elements *state* (lines 2 and 10), *city* (lines 3 and 11), and *zip* (lines 4 and 12), defined in both the schemas *address* and *LatLongReturn*, record the state, city, and zip information, respectively. In addition, to indicate whether or not a city belongs to any state, it uses the schemas *Municipality* and

City to define the elements *state* and *city*, where the element *city* may also be a child node of the element *state* (type: Municipality) in line 21.

We give a scenario that reveals a fault in the application. Ziyi Zhang, living in the city *HuangShan*, wants to find the DSL service status of her city. Hence, she inputs her name for enquiry. By searching the database with the given input, the name *HuangShan* is retrieved. The service *City2GeoSerivce* then finds out the corresponding zip information of *HuangShan*. Finally, through the zip code, the service *ServiceAvailable* provides the DSL service status of *HuangShan*.

In fact, there are two cities called *HuangShan* in Anhui, China. For the ease of discussion, we refer to them as $HuangShan_A$ and $HuangShan_B$. Although Ziyi lives in $HuangShan_A$, she may obtain the DSL service status of $HuangShan_B$ instead, because *City2GeoSerivce* merely uses the name of a city as the input to the zip information query. For *HuangShan*, it may return either of the two zip codes for $HuangShan_A$ and $HuangShan_B$, and hence the XPath may select the wrong one and assign it to *ZipCode*.

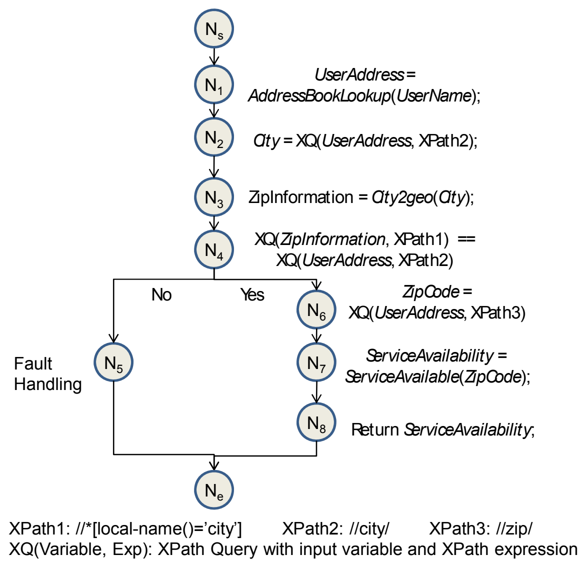
```
1    <xsd:complexType name="address">
2       <xsd:element name="state" type="xsd:Municipality"/>
3       <xsd:element name="city" type="xsd:City"/>
4       <xsd:element name="zip" type="xsd:string" />
5       <xsd:element name="StreetNum" type="xsd:int"/>
6       <xsd:element name="StreetName" type="xsd:string"/>
7       <xsd:element name="County" type="xsd:string" />
8    </xsd:complexType>
9    <xsd:complexType name="LatLongReturn">
10      <xsd:element name="state" type="xsd:Municipality "/>
11      <xsd:element name="city" type="xsd: City"/>
12      <xsd:element name="zip" type="xsd:string" />
13      <xsd:element name="County" type="xsd:string" />
14      <xsd:element name="FromLongitude" type="xsd:decimal"/>
15      <xsd:element name="FromLatitude" type="xsd:decimal"/>
16      <xsd:element name="ToLongitude" type="xsd:decimal" />
17      <xsd:element name="ToLatitude" type="xsd:decimal" />
18   </xsd:complexType>
19   <xsd:complexType name="Municipality" >
20      <xsd:element name="name" type="xsd:string"/>
21      <xsd:element name="city" type="xsd:City"/>
22   </xsd:complexType>
23   <xsd:simpleType name="City" typle="xsd:string"/>
```

Figure 2. XML Schemas for *address* and *LatLongReturn*

Intuitively, good application systems may provide a list of cities for users to choose under such a situation. However, given the application in the motivating example, and without revealing a relevant failure, it is difficult to identify the fault in the first place.

Following [28], the business process in Figure 1 can be modeled as a control flow graph (CFG), as shown in Figure 3. The mapping between the two figures is omitted. In Figure 3, we use a more concise notation *XQ* to represent XPath_Query in Figure 1.

In Figure 3, the XPath `//city/` searches the targeted city based on the variable *UserAddress*. According to the *address* schema in Figure 2, some cities (such as Hong Kong and Beijing) may not belong to any state, whereas other cities may belong to some states. Two *conceptual* paths `/state/city/` and `/city/` may reach a city field in an address. Figure 4 shows four scenarios with different contents in *UserAddress*.



N_s
N_1
UserAddress = *AddressBookLookup*(*UserName*);
N_2
City = XQ(*UserAddress*, XPath2);
N_3
ZipInformation = *City2geo*(*City*);
N_4
XQ(*ZipInformation*, XPath1) == XQ(*UserAddress*, XPath2)
No
Yes
N_6
ZipCode = XQ(*UserAddress*, XPath3)
Fault Handling
N_5
N_7
ServiceAvailability = *ServiceAvailable*(*ZipCode*);
N_8
Return *ServiceAvailability*;
N_e
XPath1: //*[local-name()='city']   XPath2: //city/   XPath3: //zip/
XQ(Variable, Exp): XPath Query with input variable and XPath expression

Figure 3. CFG for Business Process *IsServiceAvailable*

<address>	<address>	<address>	<address>
<state>	<state />	<state>	<state />
<name>Beijing</name>	<city>Beijing</city>	<name>Beijing</name>	<city />
<city>Beijing</city>	<zip>10002</zip>	<city />	<zip />
</state>	……	</state>	……
<city>Beijing</city>	</address >	<city />	</address>
<zip>10001</zip>		<zip>10003</zip>	
……		……	
</address >		</address>	
Scenario 1	Scenario 2	Scenario 3	Scenario 4

Figure 4. Scenarios for *XQ*(*UserAddress*, //city/)

For scenario 1, either `/state/city/` or `/city/` returns "Beijing" as the city. For scenarios 3 and 4, both `/state/city/` and `/city/` return no result. For scenario 2, if `/state/city/` is used, we obtain no result, but if `/city/` is used instead, the value "Beijing" will be returned. They are different.

We further study how the procedure of XPath affects the execution of workflow steps. Suppose there are three records for Beijing, as in scenarios 1 to 3. Considering the following two cases: (i) For scenarios 2 and 3, both *XQ*(*UserAddress*, XPath2) and *XQ*(*ZipInformation*, XPath1) may either return "Beijing" or no result, and hence the predicate at N_4 is not decidable. (ii) For scenarios 1 and 2, even when the predicate at N_4 is satisfied, if scenario 1 is used, the zip code will be 10001, and if scenario 2 is used, it will be 10002. This is an anomaly. It will pose an integrated problem if a follow-up service, e.g., *ServiceAvailable*, only uses the zip code to determine the availability of the DSL service. Suppose 10001 is the correct zip code for "Beijing" while 10002 is wrong. Then *ServiceAvailable* will return correct information under scenario 1 but will fail under scenario 2.

The testing challenge illustrated by the motivating example is that XPath may retrieve different data from XML messages according to XPath expressions as well as the structure of the XML schema. However, the interactions between these two types of artifact are not coded explicitly in a WS-BPEL application. In the next section, we review the preliminaries of WS-BPEL. Then, in Sections 3 and 4, we present our proposal to address the issue.

2.2 Fundamentals

We use WS-BPEL (previously known as BPEL4WS) [1] in this paper. Three critical parts in WS-BPEL are BPEL, XPath, and web services. In this paper, we focus on the interactions between XPath and BPEL, and treat web services as external services. The testing of web services is not within the scope of the paper. We introduce XPath in this section.

We adopt the definition of XPath expressions in [21]. Thus, an *XPath expression* is defined recursively using the following grammar:

$$q \rightarrow n \mid * \mid . \mid q / q \mid q // q \mid q[q]$$

where $n \in \sum$ is any label, "*" denotes a wildcard label, and "." (the dot symbol) denotes the current node. The constructs / and // mean child and descendant navigations, respectively, and [] denotes a predicate. The symbols in $\sum$ represent element labels, attribute labels, and text values that can occur in XML documents. The set of all trees are denoted by $T_{\sum}$, and each tree represents an XML document satisfying an XML schema Ω. To simplify the presentation, we also use Ω to represent the set of labels that can occur in the XML schema Ω. An XML schema is also an XML document. An *XPath Query* $q(t)$, which performs a query on a tree $t \in T_{\sum}$ using an XPath expression, returns a set of nodes in t. For a tree $t \in T_{\sum}$, $NODES(t)$ and $EDGES(t)$ denote the sets of nodes and edges, respectively. $LABEL(x)$ is the label at node x, and $LABEL(x) \in \sum$. $EDGES^*(t)$ denotes the reflexive and transitive closure of $EDGES(t)$. By induction on the structure of q, reference [21] gives the following definitions to represent a fragment of XPath, and we label them as Rules 1 to 6. According to [21], this provides a representative XPath fragment sufficient as a basis for the study of XPath.

			Rule
$n(x)$	=	$\{y \mid (x, y) \in EDGES(t), LABEL(y) = n\}$	… 1
$*(x)$	=	$\{y \mid (x, y) \in EDGES(t)\}$	… 2
$.(x)$	=	$\{x\}$	… 3
$(q_1/q_2)(x)$	=	$\{z \mid y \in q_1(x), z \in q_2(y)\}$	… 4
$(q_1//q_2)(x)$	=	$\{z \mid y \in q_1(x), (y, u) \in EDGES^*(t), z \in q_2(u)\}$	… 5
$(q_1[q_2])(x)$	=	$\{y \mid y \in q_1(x), q_2(y) \neq \emptyset\}$	… 6

Figure 5. Syntax of a Representative Fragment of XPath [21]

Each definition in Figure 5 is of the form *left = right*. We treat these definitions as left-to-right rewriting rules [6]. We further group these rules into two categories according to whether a rule can be recursively defined using other rules: A-Rules (namely Rules 1, 2, and 3) and C-Rules (Rules 4, 5, and 6), representing atomic and complex XPaths, respectively. If an XPath expression q is a C-Rule expression, it may be rewritten into a composition of multiple sub-terms, each of which is an A-Rule, a C-Rule, or an atomic relation (such as $\{(y, u) \in EDGES^*(t)\}$ in Rule 5). For instance, the XPath expression //city/ can be considered as *//city/* or just *//city/ in practice. If q is an A-Rule, it cannot be further rewritten using other rules. To ease our discussion, we refer to $q_1(x)$ and $q_2(y)$ in Rules 4, 5 and 6 as the *left* and *right* sub-terms, respectively, of the rule. For Rule 5, besides the *left* and *right* sub-terms $q_1(x)$ and $q_2(u)$, there is also a *middle* sub-term $\{u \mid (y, u) \in EDGES^*(t)\}$, which means all the nodes u in t reachable from y. We define Rule 7 as

$$//(x) = \{y \mid (x, y) \in EDGES^*(t)\} \qquad \text{… Rule 7}$$

Rule 7 is also an A-Rule. Since it is only used in Rule 5, we do not process it like other rules.

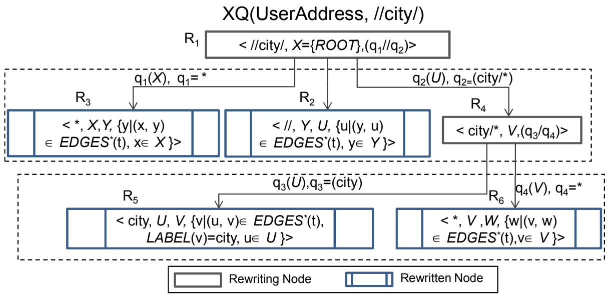
3. OUR MODEL FOR WS-BPEL

This section proposes our effort, using the *X-WSBPEL* model, to capture the interactions between BPEL and XPath.

3.1 XPath Rewriting Graph (XRG)

In the motivating example, we have illustrated that different paths taken by an XPath may result in integration problems to WS-BPEL applications. In this section, we propose an *XPath Rewriting Graph* (*XRG*), which forms an explicit artifact to represent different paths conceptually defined in an XPath expression over a schema Ω.

An XPath expression over a schema Ω can be conceptually rewritten into another form (Section 2.2) by using Rules 1 to 6. Following the nature of rewriting rules, we model such a rewriting step as a directed edge (a, b) of a graph, which links up a node a that the rewriting rule will be applied to, and a node b that represents the result after applying the rule.



XQ(UserAddress, //city/)
R1
< //city/, X={ROOT},(q1//q2)>
q1(X), q1=*
q2(U), q2=(city/*)
R3
R2
R4
< *, X,Y, {y|(x, y) ∈ EDGES*(t), x∈ X }>
< //, Y, U, {u|(y, u) ∈ EDGES*(t), y∈ Y }>
< city/*, V,(q3/q4)>
q3(U),q3=(city)
q4(V), q4=*
R5
R6
< city, U, V, {v|(u, v)∈ EDGES*(t), LABEL(v)=city, u∈ U }>
< *, V ,W, {w|(v, w) ∈ EDGES*(t),v∈ V }>
Rewriting Node
Rewritten Node

Figure 6. Example of XPath Rewriting Graph

Thus, there are two types of node in our model, as illustrated in Figure 6: (i) *Rewriting node* ⟨q, L^c, *rule*⟩, where q is a query expression; L^c ($\subseteq NODES(\Omega)$) is the current set of nodes in Ω located by the previous query step; and *rule* denotes the rewriting rule used to generate the sub-terms in this node. Initially, L^c is assigned to $\{ROOT\}$, where $ROOT$ is the root node of Ω. (ii) *Rewritten node* ⟨q, L^c, L^n, S⟩, where q and L^c carry the same meaning as in *rewriting node*; L^n ($\subseteq NODES(\Omega)$) 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 (directly obtained according to the rules in Figure 5).

For an XML document satisfying Ω (say, a returned message from a web service), L^c of a rewriting node or L^n of a rewritten node represents a set of tags, relevant to a query q, that may appear in the XML document. In Figure 4, for instance, the tag of the value "10001" is "zip", which will be an element in L^c or L^n, depending on the given XPath query. Applying Rule 5, we obtain S as $\{z \mid y \in q_1(x), (y, u) \in EDGES^*(t), z \in q_2(u)\}$, in which u, x, y, and z are called *conceptual variables*.

The definition for XRG is given in Definition 1. It is followed by an algorithm to construct an XRG.

Definition 1 (XPath Rewriting Graph) An *XPath Rewriting Graph* (*XRG*) for an XPath Query is a 5-tuple ⟨q, Ω, N_x, E_x, V_x⟩:

(a) q is an XPath expression for the XPath Query, and Ω is an XML schema that describes the XML document to be queried on.

(b) 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.

(c) 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* is used to construct an XRG. It 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. X is initially assigned as a singleton set containing the root of the schema [21]. The query q starts with this value of X to search for other nodes.

The following auxiliary functions are used: (i) Function *Root*(*XRG*) returns the root node of *XRG*. (ii) Function *match*(q, p) returns true when q can be rewritten in the form specified by the pattern p, and returns false otherwise. (iii) Function *getLastNode*(Z) returns the last rewritten node in Z, and function *getLastChild*(n, Z) returns the last rewritten child node of n in Z, both using standard *inorder traversal* [14]; (iv) Functions *setLeftChild*(n,n_l), *setMidChild*(n,n_m), and *setRightChildNode*(n,n_r) set n_l, n_m, and n_r as the *left*, *middle*, and *right* child nodes for n, respectively. As we shall explain in Section 4, the left, middle, and right child labels are important for identifying the conceptual paths of an XPath expression over Ω. In our algorithm, each label in a set is associated with the variable that locates it. (v) Functions *LabelVarDef*(X, x, n) and *LabelVarUse*(X, x, n) mark the variable x as the definition and use occurrences at node n, respectively, and associate such occurrences of the variable to every element in X. In addition, we define "x∘y" as the attribute y of x.

```
1   Global Variables $N_x$, $E_x$, $V_x$. /* Initially, $N_x \leftarrow \emptyset$, $E_x \leftarrow \emptyset$, $V_x \leftarrow \emptyset$. */
2   Algorithm   Compute_XRG
3   Input       XPath expression $q$, XML schema $\Omega$,
                Set of located nodes $X$
4   Output      $XRG$
5   let $XRG$ be $\langle q, \Omega, N_x, E_x, V_x \rangle$.
    /* Process A-Rule (rules that cannot be further rewritten) */
    /* Process Rule 1 */
6   if match($q$, "$n$") {
7       New variables $x, y$.   $V_x \leftarrow V_x \cup \{x, y\}$.
8       $Y \leftarrow \{y \mid (x, y) \in EDGES(\Omega) \wedge LABEL(y) = q, x \in X\}$.
9       $n \leftarrow \langle q, X, Y, \{y \mid (x, y) \in EDGES(\Omega) \wedge LABEL(y) = q\} \rangle$.
10      $LabelVarUse(X, x, n)$.   $LabelVarDef(Y, y, n)$.
11      $N_x \leftarrow N_x \cup \{n\}$.   /* New rewritten node */
12  }
    /* Process Rule 2 */
13  else if match($q$, "*") {
14      New variables $x, y$.   $V_x \leftarrow V_x \cup \{x, y\}$.
15      $Y \leftarrow \{y \mid (x, y) \in EDGES(\Omega), x \in X\}$.
16      $n \leftarrow \langle q, X, Y, \{y \mid (x, y) \in EDGES(\Omega)\} \rangle$. /* New rewritten node */
17      $LabelVarUse(X, x, n)$.   $LabelVarDef(Y, y, n)$.
18      $N_x \leftarrow N_x \cup \{n\}$.
19  }
    /* Process Rule 3 */
20  else if match($q$, ".") {
21      $n \leftarrow \langle q, X, X, \{x \mid x \in X\} \rangle$.   /* New Rewritten Node */
22      $LabelVarUse(X, x, n)$.   $LabelVarDef(X, x, n)$.
23      $V_x \leftarrow V_x \cup \{x\}$.   $N_x \leftarrow N_x \cup \{n\}$.
24  }
    /* Process C-Rule (rules that can be further rewritten) */
    /* Process Rule 4 */
25  if $match$($q$, "$q_1/q_2$") {
26      New variable $y, z$.   $V_x \leftarrow V_x \cup \{y, z\}$.
27      $n \leftarrow \langle q, X, Rule\ 4 \rangle$.   /* New rewriting node */
28      if ($n_l \leftarrow CheckRecursion(q_1, X, N_x)$) =$\emptyset$ then {
29          $XRG_1 \leftarrow Compute\_XRG(q_1, \Omega, X)$.
30          $n_l \leftarrow Root\ (XRG_1)$.   $n_{last} \leftarrow getLastNode(XRG_1)$.
31          $E_x \leftarrow E_x \cup XRG_1 \circ E$.   $N_x \leftarrow N_x \cup XRG_1 \circ N$.   $V_x \leftarrow V_x \cup XRG_1 \circ V$. }
32      else { $n_{last} \leftarrow getLastChild(n_l, XRG)$. }
33      let $Y$ be $n_{last} \circ L^n$.
34      if ($n_r \leftarrow CheckRecursion(q_2, Y, N_x)$) =$\emptyset$ then {
35          $XRG_2 \leftarrow Compute\_XRG(q_2, \Omega, Y)$.
36          $n_r \leftarrow Root\ (XRG_2)$.
37          $E_x \leftarrow E_x \cup XRG_2 \circ E$.   $N_x \leftarrow N_x \cup XRG_2 \circ N$.   $V_x \leftarrow V_x \cup XRG_2 \circ V$. }
38      $LabelVarUse(X, x, n_l)$.   $LabelVarDef(Y, y, n_l)$.
        $LabelVarUse(Y, y, n_r)$.   $LabelVarDef(Z, z, n_r)$.
39      $setLeftChild(n, n_l)$.   $setRightChild(n, n_r)$.
40      $E_x \leftarrow E_x \cup \{(n, n_l), (n, n_r)\}$.   $N_x \leftarrow N_x \cup \{n, n_l, n_r\}$.
41  }
    /* Process Rule 5 */
42  else if match($q$, "$q_1//q_2$") {
43      New variable $y, u, z$.   $V_x \leftarrow V_x \cup \{y, u, z\}$.
44      $n \leftarrow \langle q, X, Rule\ 5 \rangle$. /* New rewriting node */
45      if ($n_l \leftarrow CheckRecursion(q_1, X, N_x)$) = $\emptyset$ then {
46          $XRG_1 \leftarrow Compute\_XRG(q_1, \Omega, X)$.
47          $n_l \leftarrow Root(XRG_1)$.   $n_{last} \leftarrow getLastNode(XRG_1)$.
48          $E_x \leftarrow E_x \cup XRG_1 \circ E$.   $N_x \leftarrow N_x \cup XRG_1 \circ N$.   $V_x \leftarrow V_x \cup XRG_1 \circ V$. }
49      else { $n_{last} \leftarrow getLastChild(n_l, XRG)$. }
50      let $Y$ be $n_{last} \circ L^n$.
51      $U \leftarrow \{u \mid (y, u) \in EDGES^*(\Omega), y \in Y\}$.
52      $n_m \leftarrow \langle$"//", $Y, U, \{u \mid (y, u) \in EDGES^*(\Omega), y \in Y\} \rangle$.
            /* New rewritten node */
53      if ($n_r \leftarrow CheckRecursion(q_2, U, N_x)$) =$\emptyset$ {
54          $XRG_2 \leftarrow Compute\_XRG(q_2, \Omega, U)$.
55          $n_r \leftarrow Root(XRG_2)$.
56          $E_x \leftarrow E_x \cup XRG_2 \circ E$.   $N_x \leftarrow N_x \cup XRG_2 \circ N$.   $V_x \leftarrow V_x \cup XRG_2 \circ V$. }
57      $LabelVarUse(X,x,n_l)$. $LabelVarDef(Y,y,n_l)$. $LabelVarUse(Y,y,n_m)$.
        $LabelVarDef(U,u,n_m)$. $LabelVarUse(U,u,n_r)$. $LabelVarDef(Z,z,n_r)$.
58      $setLeftChild(n, n_l)$.   $setMidChild(n, n_m)$.   $setRightChild(n, n_r)$.
59      $E_x \leftarrow E_x \cup \{(n, n_m), (n, n_l), (n, n_r)\}$.
60      $N_x \leftarrow N_x \cup \{n, n_m, n_l, n_r\}$.
61  }
    /* Process Rule 6 */
62  else if match($q$, "$q_1[q_2]$") then {
63      New variable $y, z$.   $V_x \leftarrow V_x \cup \{y, z\}$.
64      $n \leftarrow \langle q, X, Rule\ 6 \rangle$.   /* New rewriting node */
65      if ($n_r \leftarrow CheckRecursion(q_1, X, N_x)$) = $\emptyset$ then {
66          $XRG_1 \leftarrow Compute\_XRG(q_1, \Omega, X)$.
67          $n_r \leftarrow Root(XRG_1)$.   $n_{last1} \leftarrow getLastNode(XRG_1)$.
68          $E_x \leftarrow E_x \cup XRG_1 \circ E$.   $N_x \leftarrow N_x \cup XRG_1 \circ N$.   $V_x \leftarrow V_x \cup XRG_1 \circ V$. }
69      else { $n_{last1} \leftarrow getLastChild(n_r, XRG)$. }
70      let $Y$ be $n_{last1} \circ L^n$.
71      if ($n_l \leftarrow CheckRecursion(q_2, Y, N_x)$) = then {
72          $XRG_2 \leftarrow Compute\_XRG(q_2, \Omega, X)$.
73          $n_l \leftarrow Root(XRG_2)$.   $n_{last2} \leftarrow getLastNode(XRG_2)$.
74          $E_x \leftarrow E_x \cup XRG_2 \circ E$.   $N_x \leftarrow N_x \cup XRG_2 \circ N$.   $V_x \leftarrow V_x \cup XRG_2 \circ V$. }
75      else { $n_{last2} \leftarrow getLastChild(n_l, XRG)$. }
76      let $Z$ be $n_{last2} \circ L^n$.
77      $Y \leftarrow Y - \{y \mid \nexists z \in Z, (y, z) \in EDGES^*(\Omega), y \in Y\}$.
78      $LabelVarUse(X, x, n_r)$.   $LabelVarDef(Y, y, n_r)$.
        $LabelVarDef(Z, z, n_l)$.   $LabelVarUse(Z, z, n_l)$.
79      $setLeftChild(n, n_l)$.   $setRightChild(n, n_r)$.
80      $E_x \leftarrow E_x \cup \{(n, n_l), (n, n_r)\}$.
81      $N_x \leftarrow N_x \cup \{n, n_l, n_r\}$.
82  }
```

```
83 return XRG.   /* XRG is finally returned */
84 Function CheckRecursion(q, L, N_x)
85     if ∃n∈N_x, q = noq ∧ L⊆noL^c then { return n. }
86     else { return Ø. }
```

The algorithm *Compute_XRG* processes Rules 1 to 6 in lines 6–12, 13–19, 20–24, 25–41, 42–61, and 62–83, respectively. Since Rules 1, 2, and 3 are A-Rules, only one rewritten node n is created in each case (lines 9, 16 and 21). Rules 4, 5 and 6 are more complex. We use Rule 4 as an example to illustrate the processing. Rules 5 and 6 are processed similarly.

Rule 4 is processed as follows: Firstly, a rewriting node n is created (line 27). Then, the algorithm recursively processes q_1 and q_2. Since there may be recursions when rewriting a node, we check the occurrence of recursions using the function *CheckRecursion* (lines 84–86) for both q_1 (lines 28–32) and q_2 (lines 34–37). If there is a node n_l or n_r for q_1 or q_2 (lines 28 and 34) involving recursions, then n_l or n_r is set as the *left* or *right* child node for n (line 39). If there is no recursion for q_1 or q_2, then the algorithm generates XRG_1 and XRG_2 (lines 29 and 35), and the root nodes of XRG_1 and XRG_2 (lines 30 and 36) will be denoted as n_l and n_r and are set as the *left* and *right* child nodes of n (line 39). Note that the input parameter X to compute XRG_2 comes from the output of the computation of XRG_1. The algorithm uses the function *getLastNode* to find the last rewritten node n_{last} in XRG_1 (line 30) when there is no recursion for q_1; and uses the function *getLastChild* to find the last rewritten child node n_{last} in XRG (line 32) when there is a recursion. n_{last} is assigned as $\langle q_{\text{last}}, L^c_{\text{last}}, Y, S^n_{\text{last}}\rangle$, and Y denotes the set of nodes used as an input parameter for constructing XRG_2. We associate each label of X with the variable definition of x at node n_l (line 38). Other variable definitions and usages are processed similarly. Hence, the XRG for an XPath Query satisfying Rule 4 is generated.

We also use *XQ*(*UserAddress*, //city/) at N_2 in Figure 3 to illustrate the algorithm. The output of the algorithm is depicted in Figure 6. To ease readers' understanding, we annotate the edges with rewriting sub-terms in Figure 6. *XQ*(*UserAddress*, //city/) is firstly identified by Rule 5 (q_1=* and q_2=city/*) (line 42), and hence rewriting node R_1 is generated (as n in line 44). Next, the algorithm recursively processes three sub-terms: //, *, and city/*. The *middle* sub-term // matches Rule 7, but the conceptual variables have been discovered by processing q_1 and q_2, and so R_2 is generated (as n_m in line 52). The *left* sub-term * matches Rule 2, and hence rewritten node R_3 is generated (as n in line 16). The *right* sub-term city/* matches Rule 4, and therefore rewriting node R_4 is generated (as n in line 27). R_4 is further rewritten into R_5 (as n_l in line 28 or 30) and R_6 (as n_r in line 34 or 36). R_5 and R_6 are the *left* and *right* children nodes of R_4, respectively. R_5 and R_6, which match Rule 1 and 2, respectively, are both rewritten nodes. Since there is no recursion for //, *, and city/*, R_2, R_3 (the root node of XRG_1, line 47), and R_4 (the root node of XRG_2, line 55) are set to be the *middle*, *left*, and *right* child nodes of R_1 (line 58), respectively. We thus finish constructing the required XRG.

We can also use Figure 6 to illustrate Definition 1. The XRG $\langle q, \Omega, N_x, E_x, V_x\rangle$ for *XQ*(*UserAddress*, //city/) is as follows: q is //city/, and Ω is the schema *address* (Figure 2) for the variable *UserAddress*. V_x is constructed by the algorithm directly, and $V_x = \{x, y, u, v, w\}$. N_x is $\{R_1, R_2, R_3, R_4, R_5, R_6\}$. E_x is $\{(R_1, R_2), (R_1, R_3), (R_1, R_4), (R_4, R_5), (R_4, R_6)\}$. The rewriting nodes are R_1 and R_4 while the rewritten nodes are R_2, R_3, R_5, and R_6.

Let us further use Figure 6 to illustrate how we handle the two paths (/state/city/ and /city/) of //city/ in the XRG. We first obtain {state, city}$\subseteq Y$ ($Y \in R_3$) from schema *address*. Then, if the value of y in R_2 is *state*, u will be *city*; whereas if the value of y is *city*, u will be undefined. Since u is the child node of y in the schema, we thus obtain the two paths /state/city/ and /city/.

We note that such rewriting can be stopped at some upper bound (say, for a huge XPath or for fragments that are not decidable). This tracks XPath at a conceptual level in a stepwise and hierarchical fashion.

3.2 X-WSBPEL Model

The structure of an XPath is denoted by an *XPath Rewriting Graph* in our model. We associate each control flow graph [8] of a BPEL program (e.g., the CFG for *IsServiceAvailable* in Figure 3) with a set of XRGs to represent a WS-BPEL application.

Definition 2 (X-WSBPEL Model) An *X-WSBPEL Model* is a couple $\langle CFG_B, XPATH\rangle$ such that

(a) CFG_B is a control flow graph representing a BPEL program P; $CFG_B = \langle N_b, E_b, V_b, s_b, e_b\rangle$, where: N_b is a set of nodes that represent the program nodes of P; E_b is a set of edges that represent the transitions between two nodes, V_b is a set of variables defined or used in BPEL; s_b is the entry node of P, and e_b is the exit node of P, $s_b, e_b \in N$.

(b) $XPATH$ is a set of XPath Rewriting Graphs denoting the occurrences of XPath in CFG_B. □

We use Figure 3 to illustrate CFG_B in the X-WSBPEL model of *IsServiceAvailable*: s_b is N_s; e_b is N_e; V_b = {*UserAddress*, *UserName*, *City*, *ZipInformation*, *ZipCode*, *ServiceAvailablity*}; $N_x = \{N_s, N_e, N_1, N_1, N_2, N_3, N_4, N_5, N_6, N_7, N_8\}$; and $E_b = \{(N_s, N_1), (N_1, N_2), (N_2, N_3), (N_3, N_4), (N_4, N_5), (N_4, N_6), (N_6, N_7), (N_7, N_8), (N_5, N_e), (N_8, N_e)\}$. We also have $XPATH = \{XRG_{\text{XPath1}}, XRG_{\text{XPath2}}, XRG_{\text{XPath3}}\}$, where XRG_{XPath2} means the XRG for XPath2 //city/ (such as Figure 6), and XRG_{XPath1} and XRG_{XPath3} are interpreted similarly. In *X-WSBPEL*, we assume that either CFG_B or XRG starts with a unique entry node and ends at a unique exit node.

4. DATA FLOW ENTITIES & CRITERIA

4.1 Data Flow Associations for WS-BPEL

4.1.1 Conventional Data flow Associations

This section recalls the data flow definitions from [8][20]. A CFG is a couple (V, E), where V is a set of nodes representing statements in a program unit and E is a set of directed edges representing the transitions among statements. A *complete path* in a CFG is a path starting from the entry node and ending with an exit node. A variable x is *defined* or has a *definition* occurrence at node n if the value of x is stored or updated at n. A variable x is *used* or has a *use* occurrence at n if the value of x is fetched or referenced at n. A *sub-path* $\langle n_i, \ldots, n_j\rangle$ is said to be *definition-clear* with respect to the variable x when none of $n_i, \ldots, n_j$ 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 there is a *definition-clear* sub-path (possibly empty) with respect to x from n_d to n_u, exclusively.

In an *X-WSBPEL* model, a WS-BPEL application consists of a CFG_B associated with a set of XRGs. Def-use associations on CFG_B in the *X-WSBPEL* model can be identified in the same style as [8]. In Figure 3, for instance, the variable *City* has a definition occurrence at node N_2 and a usage occurrence at N_3. Since there is no definition between N_2 and N_3, the path from N_2 to N_3 is definition-clear, and hence there is a def-use association for the variable *City*, denoted by $\langle City, N_2, N_3\rangle$.

4.1.2 Conceptual Paths in XRG

In the *Compute_XRG* algorithm, we have explicitly marked some child nodes as left child, right child, and middle child. This marking is important. For example, let us consider the children nodes (R_2, R_3, and R_4) of R_1 in Figure 6. The output set Y, which is defined by the variable y at R_3, will be used at R_2 as a part of a condition to define U. The set U is used by the child graph (actually R_5) with R_4 as the root. The traversal of the graph is important; otherwise, a proper conceptual relationship among variables cannot be obtained.

To apply data flow analysis and testing to an XRG, we should, therefore, respect such ordering of nodes; otherwise one may construct illegitimate data flow associations or miss legitimate ones [8]. We have designed the *Compute_XRG* algorithm to support the *inorder traversal* algorithm of [14] for constructing the path sets of a given XRG so that path sets can be treated as if they were paths in the CFG of a program unit [8]. We note, however, that a path in an *XPath Rewriting Graph* is only a model of an XPath and will never be executed by any actual program. Hence, we call them *conceptual paths*. Despite such philosophical difference, data flow associations [8] can be computed from a CFG transformed from an XRG based on the *inorder traversal* algorithm, where nodes of the CFG are nodes of the XRG, and there is an edge (a, b) in the CFG if (a, b) is a subsequence of a conceptual path of the XRG.

4.1.3 Special Handling for XRG

This section discusses def-use associations in XRGs. In an XRG, a rewriting node is used to identify a rewriting rule, where any left, right, or middle sub-term of an XPath expression will be rewritten to one or more rewritten nodes. Hence, a rewritten node contains the rule matching information. Also, since every rewriting node contains no variable definition or usage, we choose to hide them in the CFG constructed from an XRG. For instance, Figure 7 shows an example path of Figure 6 obtained by *inorder traversal* starting from R_1 without showing any rewriting nodes. One may observe that, in such a conceptual path, the variables on each node are captured in set-theoretic notation. Also, every label on a rewritten node is marked as a definition, usage, or both. At run-time, when the set L^n of a preceding node (e.g., Y in R_3) is empty, the variables in the succeeding node (e.g., R_2) as well as the variable y at R_3 will be undefined. When the XPath query is completed, it will assign values (probably empty in this case) to N_2 for the BPEL program. Hence, a path in Figure 7 actually represents 4 paths that may be taken by an XPath Query at runtime: $\langle R_3, N_2\rangle$, $\langle R_3, R_2, N_2\rangle$, $\langle R_3, R_2, R_5, N_2\rangle$, $\langle R_3, R_2, R_5, R_6, N_2\rangle$. In other words, there are *implicit predicates* in the *conceptual path* to decide the legitimate path to be taken. In Figure 7, if no element in the XML document can be selected as "y" in R_3, the set Y will be empty. This will result in the selection of path $\langle R_3, N_2\rangle$.

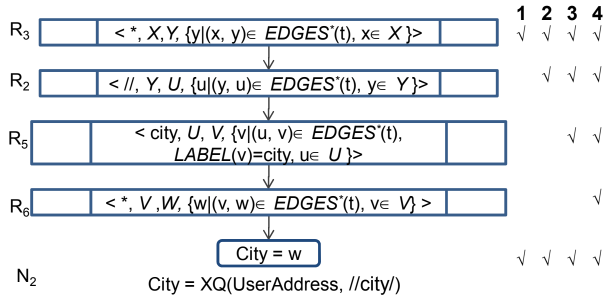

1 2 3 4
R3
< *, X,Y, {y|(x, y)∈ EDGES*(t), x∈ X }>
√ √ √ √
R2
< //, Y, U, {u|(y, u)∈ EDGES*(t), y∈ Y }>
√ √ √
R5
< city, U, V, {v|(u, v)∈ EDGES*(t), LABEL(v)=city, u∈ U }>
√ √
R6
< *, V ,W, {w|(v, w)∈ EDGES*(t), v∈ V} >
√
City = w
√ √ √ √
N2
City = XQ(UserAddress, //city/)

Figure 7. Example Conceptual Path of XRG

In a rewritten node, we treat each unique occurrence of a variable associated with the label sets L^c as a variable usage, and each unique occurrence of a variable associated with the label set L^n as a variable definition. The latter is defined through the last element S of each node in standard set-theoretic notation $\{A \mid predicate(A)\}$. In addition, any occurrence of a variable in A is a variable definition, and any occurrence of a variable in $predicate(A)$ but not in A is a variable usage. For example, the occurrence of y in R_3 is a use and the occurrence of x in R_3 is a definition. Based on the above, we present variable definitions and usages for *conceptual variables* in XRGs. Since they are related to an XPath Query at runtime, we call them *q-def* and *q-use*, respectively.

Definition 3 (Q-DEF OF VARIABLES) Given an *X-WSBPEL* $\langle CFG_B, XPATH\rangle$, a *q-def* (or def_q for short) of a variable v is either (i) an occurrence of v at node n in CFG_B such that v is assigned by the return value of the XPath Query, or (ii) a definition occurrence of v at node n of an $XRG \in XPATH$. □

For simplicity, a variable definition in an *X-WSBPEL* such that it does not satisfy Definition 3 is named as def_b. We denote the set of all def_q in a WS-BPEL application by Def_q, and the set of all def_b by Def_b. In Figure 3, for instance, the definition occurrence of the variable *ZipInformation* at N_3 is a def_b. The variable *City* is assigned by *XQ*(*UserAddress*, `//city/`) at N_2 and, according to Figure 7, *XQ*(*UserAddress*, `//city/`) returns the *conceptual variable w*. Hence, we further determine that the variable *City* is defined by w (so that it is a def_q). We also find in Figure 7 that the conceptual variable y at R_3 is defined by "$y \mid (x, y) \in EDGES^*(t)$". This definition occurrence is also a def_q.

Definition 4 (Q-USE OF VARIABLES) Given an *X-WSBPEL* $\langle CFG_B, XPATH\rangle$, a *q-use* (or use_q for short) of a variable v is either (i) an occurrence of v at node n of CFG_B such that v is the input parameter of an XPath Query in n, or (ii) a use occurrence of v at node n of an $XRG \in XPATH$. □

Similar to variable definitions, a variable usage that does not satisfy Definition 4 is named as use_b. We denote the set of all use_q in an X-WSBPEL by Use_q, and the set of all use_b by Use_b. For example, the use occurrence of the variable *UserName* at N_3 is a use_b. In Figure 7, the conceptual variable x at R_3, as used by "$y \mid (x, y) \in EDGES^*(t)$", is also a use_q. Based on the definitions of q-def and q-use, we proceed to define def-use associations in our model.

Definition 5 (QUERY-DU) A *query-def-use* (or *query-du*) association α for a variable v is a triple $\langle v, n_d, n_u\rangle$ such that v is a *q-def* at n_d and a *q-use* at n_u, and there is a definition-clear sub-path (using *inorder* traversal) with respect to v from n_d to n_u.□

We note that, by a simple and mechanical translation, the definitions and usages for an ordinary BPEL variable can be expressed using XPath. Consider the following example, in which *UserAddress.ZipCode* is assigned by *ZipOnly.ZipCode*.

```
<assign><copy>
    <from variable="UserAddress" part="ZipCode" query="."/>
    <to variable="ZipOnly" part="ZipCode"/>
</copy></assign>
```

We can use the XPath Query "." to denote the fetching of the value(s) of the variable *UserAddress.ZipCode*, that is, ".(UserAddress.ZipCode) = {UserAddress.ZipCode}". In this way, in addition to XPath expressions, our approach can be applied to other BPEL variables. We further distinguish variable occurrences in a predicate (such as Rule 6 in Section 2.2) (in the sense of *p-use* in standard terminology [8]) from the rest in the set of query-du associations. We call them *query-pu* associations.

4.2 Test Adequacy Criteria for WS-BPEL

This section proposes a set of testing criteria to measure the quality of test sets to test WS-BPEL applications.

Our first test criterion is to exercise each XRG at least once. Such an adequate test set should cover all XRGs in the WS-BPEL application under test.

Criterion 1 (ALL QUERIES) A test set T satisfies the *all-queries* criterion for an *X-WSBPEL* $\langle CFG_B, XPATH\rangle$ if and only if, for each $XRG \in XPATH$, the complete path of at least one test case $t \in T$ executes *XRG* at least once. □

Nevertheless, executing an XPath Query at least once may not evaluate all conceptual variables. In Figure 6, for instance, exercising $XQ(UserAddress$, //city/) once may not execute the definition for the variable w at R_6 because, when V at R_5 is empty, R_6 will not be evaluated. In other words, a test set that satisfies the *all-queries* criterion may not cover all the def-use associations of variables in *X-WSBPEL*. Our next criterion explores the structure of XPath. It requires a test set to cover all *query-du* associations.

Criterion 2 (ALL QUERY-DU) A test set T satisfies the *all-query-du* criterion for an *X-WSBPEL* $\langle CFG_B, XPATH\rangle$ if and only if, for each query-du association α, there is at least one test case $t \in T$ such that $def\_clear(\alpha)$ is evaluated to be true. □

As predicates are important for identifying conceptual paths in an XRG, we require a test set to cover all predicates and call the criterion *all-query-pu*. We note that *all-query-du* subsumes [8][15] *all-query-pu* because all p-uses and c-uses are evaluated in *all-query-du*. Also, any query should have at least one *query-pu* occurrence because each query may encounter a scenario in which the required value cannot be extracted by the query from an XML document. Hence, *all-query-pu* subsumes *all-queries*.

Criterion 3 (ALL QUERY-PU) A test set T satisfies the *all-query-pu* criterion for an *X-WSBPEL* $\langle CFG_B, XPATH\rangle$ if and only if, for each predicate *query-def-use* association α, if the variable usage occurs in a predicate, then for each branching of the predicate, there is at least one test case $t \in T$ that exercises the definition-clear path. □

We generically treat host programs as CFGs. Flow-based testing criteria on these programs (see [8][9][15][28]) can readily be integrated with the control flow structures or data flow entities captured by an XRG to construct other testing criteria.

5. EVALUATION

This section reports the experimentation of our proposal.

5.1 Design of Experiment

We use eight open-source WS-BPEL applications [4][24][27] to evaluate our work, as shown in the second column of Table 1. These programs are frequently used in WS-BPEL studies such as [10][18][28]. Furthermore, *LoanApproval* and *BuyBook* are the sample projects that IBM and Oracle, respectively, shipped with their BPEL modeling tools. The columns "Element" and "LOC" show the number of XML elements and lines of code of each application. We implement a tool to automate the evaluation. It reports that, in total, there are 23 XPath expressions, 87 *query-p-use*, and 209 *query-du* associations. Their breakdowns are shown in the rightmost three columns of Table 1. Although the numbers of XPath in the subject programs are small, as we shall show later, the differences in effectiveness exhibited in the experiment are already significant.

Table 1. Subject programs and their descriptive statistics

Ref.	Applications	Element	LOC	Query	Query-pu	Query-du
A	ATM [4]	94	180	3	12	35
B	BuyBook [24]	153	532	3	15	26
C	DSLService [27]	50	123	3	11	47
D	GYMLocker [4]	23	52	2	9	23
E	LoanApproval [4]	41	102	2	11	19
F	MarketPlace [4]	31	68	2	9	17
G	Purchase [4]	41	125	2	6	9
H	TripHandling [4]	94	170	6	14	33

Next, we generate different faulty versions by seeding one fault into each copy of the original subject program. To our best knowledge, these faulty versions do not exist in repositories. We (members of our research group who have experience in SOA development and are non-authors) follow [15][20] to seed faults. In total, we create 60 faulty versions.

Our tool then generates test suites for our testing criteria and for random testing [8][18]. When generating each test suite for our testing criteria, the tool randomly selects a test case from a test pool and executes a target version over the test case. The test case is added to the test suite for a testing criterion only if it improves the coverage specified by the criterion. After a number of trials, we set the process to terminate if either 100% coverage of a criterion has been attained, or an upper bound of 50 trials has been reached. (We note from the experiment that the tool has consistently achieved 100% coverage for all criteria (except the all-query-du criterion) at the termination of the process.) For each version, we repeat this process 100 times. A similar approach is adapted by [15][20]. For random testing, we randomly select a test suite whose size should be the same as the maximum number of test cases in all test suites for our testing criteria on the same program version. We choose the fault detection rate [15] as the effectiveness measure in the experimentation, which is defined as the proportion of the number of test suite that can expose the fault(s) in a version to the size of the test suite.

5.2 Data Analysis

We present the results of the experiment in this section. We first calculate the coverage percentages of the test suite on the faulty versions for the respective testing criteria. The minimal, mean and maximal coverages that have been achieved by the test suites are: *all-queries* (100%, 100%, 100%), *all-quer-pu* (100%, 100%, 100%), and *all-query-du* (94.8%, 97.7%, 100%). When a test suite cannot yield 100% coverage, we deem the outstanding coverage requirements infeasible.

We partition the 60 faults into three categories (in-BPEL, in-XPATH, and in-WSDL) according to the type of artifact that each fault resides, as shown in Table 2. Columns A–H correspond to the respective references of these applications in Table 1. There are 21 faults in BPEL programs, 21 faults in XPath expressions or XML schemas, and 18 faults WSDL documents.

Table 2. Distributions of faults

Category	A	B	C	D	E	F	G	H
in-BPEL	3	2	3	3	3	2	2	3
in-XPath	3	2	3	3	3	2	2	3
in-WSDL	2	3	2	1	2	2	3	3
Total	8	7	8	7	8	6	7	9

Table 3 summarizes the fault-detection rates of the three categories of faults and the aggregated results of the experiment. As shown in the overall section of the table, random testing exhibits the worst mean effectiveness among all four criteria. It is around 20%–24% less effective than our criteria. As expected, our *all-query-du* criterion shows the best mean fault detection ability, with a fault detection rate of 97.6% in the experiment. The *all-query-du* criterion also performs well in each category.

Table 3. Fault-detection rates of testing criteria by categories

Category	Fault-detecting rates				Box-whisker usage		
	Criterion	Min.	Mean	Max.	Median	25%	75%
in-BPEL	Random	0.286	0.843	1.000	0.910	0.730	0.980
	All-queries	0.524	0.928	1.000	0.930	0.860	1.000
	All-query-pu	0.524	0.938	1.000	0.950	0.880	1.000
	All-query-du	0.667	0.976	1.000	1.000	0.990	1.000
in-XPath	Random	0.095	0.722	1.000	0.740	0.660	0.770
	All-queries	0.524	0.932	1.000	1.000	0.870	1.000
	All-query-pu	0.524	0.937	1.000	1.000	0.880	1.000
	All-query-du	0.571	0.977	1.000	1.000	0.960	1.000
in-WSDL	Random	0.167	0.621	1.000	0.650	0.470	0.763
	All-queries	0.556	0.904	1.000	0.970	0.803	1.000
	All-query-pu	0.556	0.915	1.000	0.975	0.823	1.000
	All-query-du	0.778	0.974	1.000	1.000	1.000	1.000
Overall	Random	0.183	0.734	1.000	0.745	0.615	0.910
	All-queries	0.533	0.922	1.000	0.970	0.860	1.000
	All-query-pu	0.533	0.931	1.000	0.975	0.870	1.000
	All-query-du	0.667	0.976	1.000	1.000	0.980	1.000

The *all-query-du* criterion, taking q-def and q-use (including both computation-use and predicate-use) into account, detects more faults than the *all-query-pu* criterion in the in-XPATH category (0.977 vs. 0.937) and in the in-WSDL category (0.974 vs. 0.915). We also compare random testing with our testing criteria, as shown in the three rightmost columns in Table 3. It shows that, in terms of box-whisker standards (the median and the 25% and 75% ranges), our criteria are more effective in detecting faults specified with XPath and WSDL. The average times to construct test sets for our criteria are in the range 0.45s–1.2s, running on an Intel 2.4 GHz CPU with 512 MB memory.

To compare the effectiveness of different criteria based on comparable cost [9][20], we increase the size of every test suite for random testing so that the expanded test suite will give the same mean effectiveness as the base test suite of a compared testing criterion (or when an upper bound of 200 trials has been reached). We repeat this experiment 10 times to obtain the average result in each case. The results are listed in Table 4. Columns 2 to 4 list the average size of the expanded test suite for random testing (*X*) against that of the base test suite for our criterion (*Y*), that is *X* / *Y*. Table 4 shows that our criteria use only about 20% of the mean number of test cases for random testing to attain the same effectiveness.

Table 4. Comparable test suite sizes of various criteria

Subject	Random / all-queries	Random / all-query-pu	Random / all-query-du
Atm	2.16	2.13	1.81
BuyBook	9.84	9.66	8.91
DSLService	3.61	3.59	2.78
GYMLocker	2.38	2.37	2.18
LoanApproval	8.13	8.01	7.97
MarketPlace	13.40	13.13	12.06
Purchase	3.31	3.18	2.85
TripHanding	3.47	3.40	2.48
Average	5.13	5.05	4.29

In summary, the experimental results show that our approach detects over 90% of all faults and uses much fewer test cases than random testing to achieve the same effectiveness. In the future, we shall study how to detect subtle faults more effectively.

5.3 Threats to Validity and other Discussions

We have carefully developed a tool to perform instrumentation and collect statistical information for evaluation. We only use a limited number of programs and certain types of fault in our experiments. Like most other empirical studies, the result of our empirical study may not be generalized to cover all cases. WS-BPEL applications support program concurrency. We apply the notion of forced deterministic testing for concurrent programs to conduct the experiment.

XPath has been included in Java 6 (see javax.xml.xpath). By modeling Java as a CFG and XPath as an XRG, our approach can readily be applied to a host language with such XPath support.

One may wonder how program instrumentation can be done in BPEL programs. The following is a sample solution: the web service *instrument* is used to output the value of the variable *ZipOnly.ZipCode* and the XPath expression "." at runtime. With such instrumentation, necessary information for computing the coverage of test suite can be obtained.

```
<from variable="UserAddress" part="ZipCode" query="."/>
<to variable="ZipOnly" part="ZipCode"/>
<invoke name="instrument".variable="UserAddress.ZipCode" query="."/>
```

6. RELATED WORK

Broy and Krüger [5] study an interacting component and a service in the system as *total* behavior and *partial* behavior, respectively. They model a service having dual properties based on the notion of processes and partial functions. They do not study testing, however. Ye et al. [29] models a service as a process, and studies the encapsulation effects of actions on atomicity to support service transactions. Neither work considers XPath in services.

Modeling BPEL and web service components using a state model is popular. Mongiello and Castelluccia [22] translate a BPEL program into such a model and apply model checking to verify temporal properties. Schmidt and Stahl [25] model it using Petri nets instead. Apart from using a state model to represent a BPEL program, Foster et al. [7] further analyze and verify the interactions between BPEL programs and web services based on WS-BPEL specifications. Fu et al. [10][11] translate web services into Promela for formal verification using their tool WSAT. Their approach differs from ours. We cover different fragments of XPath, and it is unclear to us whether their fragment is decidable. Secondly, we translate an XPath strictly according to the definition of XPath expressions in [13][21], while they translate an XPath into a Promela procedural routine that uses self-proposed variables and codes to *simulate* XPath operations. Intuitively, a test suite covering the data flow associations in a translated routine would test the implementation rather than the declaration as expressed in the WS-BPEL application. Our testing approach addresses the interactions captured in various artifacts: BPEL programs, WSDL documents, XML schemas, and XPath expressions. The above work complements ours.

García-Fanjul et al. [12] treat a WS-BPEL application as a finite state machine and use mutation analysis to generate faulty versions. They then check each faulty version against a given temporal property using SPIN, and any counterexample thus generated will be treated as a test case. Yan et al. [28] model a WS-BPEL application as a set of concurrent finite state machines, use a heuristic approach to conduct reachability analysis to find concurrent paths, and use such paths as test cases.

Many previous papers [8][9][17][26] on data flow testing are based on information derivable from program statements. Although it is emerging to consider the effect of pervasive computing environment on programs [19][20], to our best knowledge, none of them explores the integration of heterogeneous sorts of technique such as program representation, dataflow analysis, and declarative semantics on diverse types of artifact as we do in this paper.

7. CONCLUSIONS

WS-BPEL applications are a type of service-oriented workflow application. In these applications, a business process is specified as a BPEL program, and individual loosely-coupled workflow steps are linked up via the exchange of XML-based messages. Failing to extract a right piece of data from an XML message, for instance, will pose an integration error in such an application. On the other hand, XML is fundamental to many service-oriented workflow applications, and XPath is the means to query on XML documents. The extensive usage of XPath poses a demand to study how to test these applications effectively.

The paper has proposed a novel approach to studying XPath at a conceptual level, developed a data structure known as XPath Rewriting Graph (XRG) to capture how an XPath can be rewritten from one form to another in a stepwise fashion, and proposed an algorithm to construct XRGs. An XRG captures the mathematical variables to support stepwise rewriting of XPath. We also use these conceptual variables as if they were program variables to determine the def-use associations in an XRG, and integrate them with the ordinary ones in a host program. We make no particular assumption about the host program and generically treat it as a control flow graph. Based on the extended variables, we propose a family of testing criteria to test WS-BPEL programs. The experimental results confirm that our proposal is promising.

Our paper demonstrates a new strategy that transforms schema-based definitions into external artifacts and then use these derived artifacts for analysis, design, and testing. This strategy is applicable to undecidable programs in general (albeit with an upper-bound setting to avoid infinite rewriting), and XPath is only one example. In the future, we shall proceed along this line to study the application of our novel approach to other forms of (un)decidable artifacts.

8. ACKNOWLEDGMENT

We thank Zhenyu Zhang of The University of Hong Kong for his insightful comments to improve the paper.

9. REFERENCES

[1] C. Barreto et al., Eds. *Web Services Business Process Execution Language Version 2.0: Primer*. OASIS, 2007. Available at http://docs.oasis-open.org/wsBPEL/2.0/wsBPEL-v2.0.html.

[2] B. Benatallah, R.M. Dijkman, M. Dumas, and Z. Maamar. Service composition: concepts, techniques, tools and trends. In *Service-Oriented Software Engineering: Challenges and Practices*, pages 48–66. Idea Group Publishing, 2005.

[3] A. Berglund et al., Eds. *XML Path Language* (*XPath*) *2.0: W3C Recommendation*. World Wide Web Consortium, 2007. Available at http://www.w3c.org/TR/xpath20/.

[4] *BPWS4J: a Platform for Creating and Executing BPEL4WS Processes*, Version 2.1. IBM, 2004. Available at http://www.alphaworks.ibm.com/tech/bpws4j.

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

[6] H.Y. Chen, T.H. Tse, F.T. Chan, and T.Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM TOSEM*, 7 (3): 250–295, 1998.

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

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

[9] P.G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of SIGSOFT '98/FSE-6*, pages 153–162. 1998.

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

[11] X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *Proceedings of ISSTA 2004*, pages 252–262. 2004.

[12] J. García-Fanjul, J. Tuya, and C. de la Riva. Generating test cases specifications for BPEL compositions of web services using SPIN. In *Proceedings of WS-MaTe 2006*, pages 83–94. 2006.

[13] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of XPath query evaluation and XML typing. *JACM*, 52 (2): 284–335, 2005.

[14] D. Gries and J.L. van de Snepscheut. Inorder traversal of a binary tree and its inversion. In *Formal Development Programs and Proofs*, pages 37–42. Addison Wesley, 1989.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of ICSE '94*, pages 191–200. 1994.

[16] C. Innocenti. SOA and the importance of XQuery. *SOA Magazine* Issue X, September 2007.

[17] G.M. Kapfhammer, and M.L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of ESEC 2003/FSE-11*, pages 98–107. 2003.

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

[19] H. Lu, W.K. Chan, and T.H. Tse. Testing pervasive software in the presence of context inconsistency resolution services. In *Proceedings of ICSE 2008*. 2008.

[20] 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.

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

[22] M. Mongiello and D. Castelluccia. Modelling and verification of BPEL business processes. In *Proceedings of MBD-MOMPES 2006*, pages 144–148. 2006.

[23] D. Olteanu, T. Furche, and F.Bry. Evaluating complex queries against XML streams with polynomial combined complexity. In *Key Technologies for Data Management*, volume 3112 of LNCS, pages 31–44. 2004.

[24] *Oracle BPEL Process Manager*. Oracle Technology Network. Available at http://www.oracle.com/technology/products/ias/bpel/.

[25] K. Schmidt and C. Stahl. A Petri net semantic for BPEL4WS: validation and application. In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets*, pages 1–6. 2004.

[26] A.L. Souter and L.L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE TSE*, 29 (11): 1005–1018, 2003.

[27] *Web Services Invocation Framework: DSL Provider Sample Application*. Apache Software Foundation. Available at http://svn.apache.org/viewvc/webservices/wsif/trunk/java/samples/dslprovider/README.html?view=co.

[28] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang. BPEL4WS unit testing: test case generation using a concurrent path analysis approach. In *Proceedings of ISSRE 2006*, pages 75–84. 2006.

[29] C. Ye, S.C. Cheung and W.K. Chan. Publishing and composition of atomicity-equivalent services for B2B collaboration. In *Proceedings of ICSE 2006*, pages 351–360. 2006.