<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Data flow testing of service-oriented workflow applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Mei, L; Chan, WK; Tse, TH</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), 2008, p. 371-380</td>
</tr>
<tr>
<td><strong>Issued Date</strong></td>
<td>2008</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10722/55467">http://hdl.handle.net/10722/55467</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.; Copyright ACM 2008. 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 International Conference on Software Engineering Proceedings. 30th international conference on Software engineering. Pages 371-380. 2008 ISBN:978-1-60558-079-1 DOI <a href="http://doi.acm.org/10.1145/1368088.1368139">http://doi.acm.org/10.1145/1368088.1368139</a></td>
</tr>
</tbody>
</table>
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.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACSF ’08, May 10–18, 2008, Leipzig, Germany.  
Copyright 2008 ACM 978-1-60558-079-1/08/05...$5.00.
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 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, XRG 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 IsServiceAvailable 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 A1, A2, …. As to ease subsequent discussions. The service IsServiceAvailable is described as follows:

(1) A1 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 City.

(2) A2 extracts the city name from Username via the XPath //city/ and assigns the city name to the BPEL variable City.

(3) A3 invokes the service City2GeoService, which looks up the zip information based on the given City, and keeps the result in the variable ZipInformation.

(4) A4 checks whether the city name in Username is the same as that in ZipInformation by extracting their city fields through the XPath //city/ and //*[local-name()='city'], respectively, where local-name() is an XPath function that returns the name of an element.

(5) If A4 detects no problem, A4 further extracts the zip code from Username via the XPath //zip/ and assigns it to the BPEL variable ZipCode. Then, A5 executes the service ServiceAvailable to obtain the service availability status, and finally A6 returns the ServiceAvailability information to the caller.

(6) If A4 detects a problem, A7 will execute a fault handler.
"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 "City2GeoService" 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 "City2GeoService" 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".

Although Ziyi lives in "HuangShan", she may obtain the DSL service status of "HuangShan_B" instead, because "City2GeoService" 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".

![Figure 2. XML Schemas for address and LatLongReturn](image)

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".

![Figure 3. CFG for Business Process IsServiceAvailable](image)

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 $XQ(\text{UserAddress}, \text{XPath2})$ and $XQ(\text{ZipInformation}, \text{XPath1})$ may either return "Beijing" or no result, and hence the predicate at $N_2$ is not decidable. (ii) For scenarios 1 and 2, even when the predicate at $N_2$ 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 integrative problem if a follow-up service, etc., "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:

\[ q \rightarrow n | q / q | q | q \]

where \( n \in \Sigma \) is any label, \( \ast \) denotes a wildcard label, and \( \cdot \) (the dot symbol) denotes the current node. The constructs \( / \) and \( \cdot / \) mean child and descendant navigations, respectively, and \( [ ] \) denotes a predicate. The symbols in \( \Sigma \) represent element labels, attribute labels, and text values that can occur in XML documents. The set of all trees are denoted by \( T_\Sigma \), and each tree represents an XML document satisfying an XML schema \( \Omega \). To simplify the presentation, we also use \( \Omega \) to represent the set of labels that can occur in the XML schema \( \Omega \). An XML schema is also an XML document. An XPath Query \( q(t) \), which performs a query on a tree \( t \in T_\Sigma \) using an XPath expression, returns a set of nodes in \( t \). For a tree \( t \in T_\Sigma \), NODES\( (t) \) and EDGES\( (t) \) denote the sets of nodes and edges, respectively. LABEL\( (x) \) is the label at node \( x \), and \( \text{LABEL}(x) = \sum \) 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 \( \text{XPath} \).

\[ \text{Rule 1: } n \rightarrow \text{Label} \rightarrow \{ \text{NODES}(t), \text{EDGES}(t) \} \]

\[ \text{Rule 2: } q \rightarrow \text{Label} \rightarrow \{ \text{NODES}(t), \text{EDGES}(t) \} \]

\[ \text{Rule 3: } q \rightarrow \{ \text{NODES}(t), \text{EDGES}(t) \} \]

\[ \text{Rule 4: } \text{Label}\{ q_1(q_2)(x) \} \rightarrow \{ \text{NODES}(t), \text{EDGES}(t) \} \]

\[ \text{Rule 5: } \text{Label}\{ q_1(q_2)(x) \} \rightarrow \{ \text{NODES}(t), \text{EDGES}(t) \} \]

\[ \text{Rule 6: } \text{Label}\{ q_1(q_2)(x) \} \rightarrow \{ \text{NODES}(t), \text{EDGES}(t) \} \]

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

Each definition in Figure 5 is of the form \( \text{left} = \text{right} \). 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 \( \text{q} \) 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 \( \text{Label}(x) = \sum \) in Rule 5). For instance, the XPath expression \( / \text{city} / \cdot \) can be considered as \( \ast / \text{city} / \ast \) or just \( \ast / \text{city} / \) in practice. If \( \text{q} \) 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 \( q_2(y) \), there is also a middle sub-term \( \{ u | (u, v) \in \text{EDGES}(t), z = q_3(u) \} \), which means all the nodes \( u \) in \( t \) reachable from \( y \). We define Rule 7 as

\[ (\text{i}) = \{ \text{NODES}(t), \text{EDGES}(t) \} \]

\[ \text{Rule 7: } (\text{i}) \rightarrow \{ \text{NODES}(t), \text{EDGES}(t) \} \]

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 \( \Omega \).

An XPath expression over a schema \( \Omega \) 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.

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 \( (q, L', \text{rule}) \), where \( q \) is a query expression; \( L' \) is the current set of nodes in \( \Omega \) located by the previous query step; and \( \text{rule} \) denotes the rewriting rule used to generate the sub-terms in this node. Initially, \( L' \) is assigned to \( \{ \text{ROOT} \} \), where \( \text{ROOT} \) is the root node of \( \Omega \). (ii) Rewritten node \( (q, L, S) \), where \( q \) and \( L' \) carry the same meaning as in rewriting node; \( L' \) denotes the set of nodes in \( \Omega \) to be located by \( q \) starting from some node in \( L' \); 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 \( \Omega \) (say, a returned message from a web service), \( L' \) of a rewriting node or \( L' \) 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 \( L' \) or \( L \), depending on the given XPath query. Applying Rule 5, we obtain \( S = \{ \text{zip}, \text{zip} \} \), \( \text{zip} \in \text{NODES}(\Omega) \), \( \text{zip} \) is an XML schema that describes the XML document to be queried on.

Definition 1 (XPath Rewriting Graph) An XPath Rewriting Graph (XRG) for an XPath Query is a 5-tuple \( (q, \Omega, N_r, E_r, V_r) \):

\[ (a) \quad q \text{ is an XPath expression for the XPath Query, and } \Omega \text{ is an XML schema that describes the XML document to be queried on.} \]

\[ (b) \quad N_r \text{ is a set of rewriting and rewritten nodes identified by the algorithm Compute_XRG, and } V_r \text{ is a set of conceptual variables defined at the nodes in } N_r. \]

\[ (c) \quad E_r \text{ is a set of edges } (x_r, y_r), \text{ each of which represents a transition from } x_r \text{ to } y_r, \text{ where } x_r \text{ is a rewriting node and } y_r \text{ is either a rewriting node or a rewritten node. All the edges are also computed by the algorithm Compute_XRG.} \]
The algorithm Compute_XRG is used to construct an XRG. It takes an XPath expression $q$, the schema $\Omega$ of some XML document, and a set of currently located nodes $X$ of $\Omega$ as parameters, and outputs the corresponding XRG. $X$ is initialized 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 $\text{Root}(XRG)$ returns the root node of $XRG$. (ii) Function $\text{match}(q, p)$ returns true when $q$ can be rewritten in the form specified by the pattern $p$, and returns false otherwise. (iii) Function $\text{getLastNode}(Z)$ returns the last rewritten node in $Z$, and function $\text{getLastChild}(n, Z)$ returns the last rewritten child node of $n$ in $Z$, both using standard inorder traversal [14]; (iv) Functions $\text{setLeftChild}(n, n_r)$, $\text{setMidChild}(n, n_m)$, and $\text{setRightChild}(n, n_r)$ set $n_r$, $n_m$, and $n$ 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 $\Omega$. In our algorithm, in each set is associated with the variable that locates it. (v) Functions $\text{LabelVarDef}(X, x, n)$ and $\text{LabelVarUse}(X, x, n)$ 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 "$\text{var}$" as the attribute $y$ of $x$.

1. **Global Variables** $N_i, E_i, V_i$. /* Initially, $N_i$ $\in$ $\Omega$, $E_i$ $\in$ $\Omega$, $V_i$ $\in$ $\Omega$. */
2. **Algorithm** Compute_XRG
3. **Input** XPath expression $q$, XML schema $\Omega$
4. **Set of currently located nodes $X$**
5. **Output** $XRG$
6. **let** $XRG$ be $(q, \Omega, N_i, E_i, V_i)$.
7. /* Process A-Rule (rules that cannot be further rewritten) */
8. /* Process Rule 1 */
9. if $\text{match}(q, \omega)$
10. 7. /* New variables $x$, $y$. */
11. $V_i \leftarrow V_i \cup \{(x, y)\}$
12. $y \leftarrow \{x, y\} \in \text{EDGES}(\Omega)$
13. $n \leftarrow \{x, y\} \in \text{LABEL}(q) \in \text{LABEL}(q)$
14. $\text{LabelVarDef}(X, x, n)$
15. $\text{LabelVarUse}(X, x, n)$
16. $N_i \leftarrow N_i \cup \{(x, y)\}$
17. /* New rewritten node */
18. /* Process Rule 2 */
19. else if $\text{match}(q, \omega')$
20. /* New variables $x$, $y$. */
21. $V_i \leftarrow V_i \cup \{(x, y)\}$
22. $XRG_1 \leftarrow XRG_1 \cup \{(x, y)\}$
23. /* New Rewritten Node */
24. $n \leftarrow \{x, y\} \in \text{LABEL}(q)$
25. $\text{LabelVarDef}(X, x, n)$
26. $\text{LabelVarUse}(X, x, n)$
27. $N_i \leftarrow N_i \cup \{(x, y)\}$
28. /* Process Rule 3 */
29. else if $\text{match}(q, \omega'')$
30. /* New variables $x$, $z$. */
31. $V_i \leftarrow V_i \cup \{(x, z)\}$
32. $XRG_2 \leftarrow XRG_2 \cup \{(x, z)\}$
33. /* New Rewritten Node */
34. $n \leftarrow \{x, z\} \in \text{LABEL}(q)$
35. $\text{LabelVarDef}(X, x, n)$
36. $\text{LabelVarUse}(X, x, n)$
37. $N_i \leftarrow N_i \cup \{(x, z)\}$
38. /* Process C-Rule (rules that can be further rewritten) */
39. /* Process Rule 4 */
40. if $\text{match}(q, \omega''')$
41. /* New variable $y$, $z$. */
42. $V_i \leftarrow V_i \cup \{(y, z)\}$
43. $XRG_3 \leftarrow XRG_3 \cup \{(y, z)\}$
44. /* New Rewriting node */
45. $n \leftarrow \{y, x\} \in \text{LABEL}(q)$
46. if $\text{match}(q, \omega''')$
47. /* New variables $x$, $y$. */
48. $V_i \leftarrow V_i \cup \{(x, y)\}$
49. $XRG_4 \leftarrow XRG_4 \cup \{(x, y)\}$
50. /* New Rewriting node */
51. $n \leftarrow \{x, y\} \in \text{LABEL}(q)$
52. if $\text{match}(q, \omega''')$
53. /* New variables $x$, $y$. */
54. $V_i \leftarrow V_i \cup \{(x, y)\}$
55. $XRG_5 \leftarrow XRG_5 \cup \{(x, y)\}$
56. /* New Rewriting node */
57. $n \leftarrow \{x, y\} \in \text{LABEL}(q)$
58. if $\text{match}(q, \omega''')$
59. /* New variables $x$, $y$. */
60. $V_i \leftarrow V_i \cup \{(x, y)\}$
61. $XRG_6 \leftarrow XRG_6 \cup \{(x, y)\}$
62. /* New Rewritten node */
63. $n \leftarrow \{x, y\} \in \text{LABEL}(q)$
64. if $\text{match}(q, \omega''')$
65. /* New variables $x$, $y$. */
66. $V_i \leftarrow V_i \cup \{(x, y)\}$
67. $XRG_7 \leftarrow XRG_7 \cup \{(x, y)\}$
68. /* New Rewritten node */
69. $n \leftarrow \{x, y\} \in \text{LABEL}(q)$
70. if $\text{match}(q, \omega''')$
71. /* New variables $x$, $y$. */
72. $V_i \leftarrow V_i \cup \{(x, y)\}$
73. $XRG_8 \leftarrow XRG_8 \cup \{(x, y)\}$
74. /* New Rewritten node */
75. $n \leftarrow \{x, y\} \in \text{LABEL}(q)$
76. if $\text{match}(q, \omega''')$
77. /* New variables $x$, $y$. */
78. $V_i \leftarrow V_i \cup \{(x, y)\}$
79. $XRG_9 \leftarrow XRG_9 \cup \{(x, y)\}$
80. /* New Rewritten node */
81. $n \leftarrow \{x, y\} \in \text{LABEL}(q)$
82. /* New Rewritten node */
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 q1 and q2. Since there may be recursions when rewriting a node, we check the occurrence of recursions using the function CheckRecursion (lines 84–86) for both q1 (lines 28–32) and q2 (lines 34–37). If there is a node n1 or n2 for q1 or q2 (lines 28 and 34) involving recursions, then n1 or n2 is set as the left or right child node for n (line 39). If there is no recursion for q1 or q2, then the algorithm generates XRG1 and XRG2 (lines 29 and 35), and the root nodes of XRG1 and XRG2 (lines 30 and 36) will be denoted as n1 and n2, and are set as the left and right child nodes of n (line 39). Note that the input parameter X to compute XRG2 comes from the output of the computation of XRG1. The algorithm uses the function getLastNode to find the last rewritten node nLast in XRG1 (line 30) when there is no recursion for q1; and uses the function getLastChild to find the last rewritten child node nLast in XRG (line 32) when there is a recursion. nLast is assigned as (qLast, Ylast, SLast), and Y denotes the set of nodes used as an input parameter for constructing XRG2. We associate each label of X with the variable definition of x at node n (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 N2 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 (q1=*, q2=city/*) (line 42), and hence rewriting node R1 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 q1 and q2, and so R1 is generated (as n in line 52). The left sub-term * matches Rule 4, and hence rewritten node R2 is generated (as n in line 16). The right sub-term city/* matches Rule 4, and therefore rewriting node R3 is generated (as n in line 27). R4 is further rewritten into R5 (as n in line 28 or 30) and R6 (as n in line 34 or 36). R5 and R6 are the left and right children nodes of Rn, respectively. R5 and R6, which match Rule 1 and 2, respectively, are both rewritten nodes. Since there is no recursion for //, *, and city/*, R2, R3, and R4 (the root node of XRG1, line 47), and R5 (the root node of XRG2, line 55) are set to be the middle, left, and right child nodes of Rn (line 58), respectively. We thus finish constructing the required XRG.

We can also use Figure 6 to illustrate Definition 1. The XRG $<q, \Omega, N_1, E_1, V_1>$ for $XQ(UserAddress, //city/*)$ is as follows: q is //city/*, and $\Omega$ is the schema address (Figure 2) for the variable UserAddress. V1 is constructed by the algorithm directly, and $V_1 = \{x, y, u, v, w\}$. N1 = \{R1, R2, R3, R4, R5, R6\}. E1 = \{(R1, R2), (R1, R3), (R1, R4), (R1, R5), (R4, R5)). The rewriting nodes are R1 and R4 while the rewritten nodes are R3, R5, R6, and Rn.

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\}зы(Y(=R3) from schema address. Then, if the value of y in R3 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 conceptual 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 \(P, s_{B}, e_{B} \in N_{B}\).

(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 \(N_{1}\); \(e_{B}\) is \(N_{2}\); \(V_{B} = \{UserAddress, UserName, City, ZipInformation, ServiceAvailability\}\); \(N_{B} = \{N_{1}, N_{2}, N_{3}, N_{4}, N_{5}, N_{6}, N_{7}, N_{8}, N_{9}\}\); and \(E_{B} = \{(N_{2}, N_{1}), (N_{1}, N_{2}), (N_{7}, N_{6}), (N_{6}, N_{5}), (N_{5}, N_{4}), (N_{4}, N_{3}), (N_{3}, N_{2}), (N_{2}, N_{1})\}\). We also have \(XPATH = \{XRG_{XSBPEL, XPath1}, XRG_{XSBPEL, XPath2}, XRG_{XSBPEL, XPath3}\}\), where \(XRG_{XSBPEL, XPath2}\) means the XRG for XPath2 //city/* (such as Figure 6), and \(XRG_{XSBPEL, XPath1}\) and \(XRG_{XSBPEL, XPath3}\) 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 \((n_0, \ldots, n_i)\) is said to be definition-clear with respect to the variable \(x\) when none of \(n_0, \ldots, n_i\) defines or undefines \(x\). A def-use association is a triple \((x, n_0, n_i)\) such that the variable \(x\) is defined at node \(n_0\) and used at node \(n_i\) and there is a definition-clear sub-path (possibly empty) with respect to \(x\) from \(n_0\) to \(n_i\) exclusively.

In an X-WSBPEL model, a BPEL-WS 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 \((City, N_{2}, N_{3})\).
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 \((R_2, R_3, \text{and } R_4)\) 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 \(R_2\)) with \(R_3\) 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 \(L^*\) of a preceding node (e.g., \(Y\) in \(R_1\)) 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: \((R_3, N_2), (R_3, R_2, N_2), (R_5, R_2, N_2), (R_5, R_2, R_3, R_4, N_2)\). 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 \((R_3, N_2)\).

![Figure 7. Example Conceptual Path of XRG](image)

In a rewritten node, we treat each unique occurrence of a variable associated with the label sets \(L^*\) as a variable usage, and each unique occurrence of a variable associated with the label set \(L'\) as a variable definition. The latter is defined through the last element \(S\) of each node in standard set-theoretic notation \(\{A\mid \text{predicate}(A)\}\). In addition, any occurrence of a variable in \(A\) is a variable definition, and any occurrence of a variable in \(\text{predicate}(A)\) but not in \(A\) is a variable usage. For example, the occurrence of \(y\) in \(R_1\) 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 \((\text{CFG}_P, \text{XPATH})\), a q-def (or \(\text{def}_v\) for short) of a variable \(v\) is either (i) an occurrence of \(v\) at node \(n\) in \(\text{CFG}_P\) 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\(\text{XPATH}\).

For simplicity, a variable definition in an X-WSBPEL such that it does not satisfy Definition 3 is named as \(\text{def}_v\). We denote the set of all \(\text{def}_v\) in a WS-BPEL application by \(\text{Def}_v\). In Figure 3, for instance, the definition occurrence of the variable ZipInformation at \(N_1\) is a \(\text{def}_v\). The variable City is assigned by \(\text{XQ}([\text{UserAddress}.//\text{city}]())\) at \(N_2\) and, according to Figure 7, \(\text{XQ}([\text{UserAddress}.//\text{city}]())\) returns the conceptual variable \(w\). Hence, we further determine that the variable City is defined by \(w\) (so that it is a \(\text{def}_v\)). We also find in Figure 7 that the conceptual variable \(y\) at \(R_3\) is defined by "\(y\mid (x,y)\in\text{EDGES}^*(y)\)". This definition occurrence is also a \(\text{def}_v\).

**Definition 4** (Q-USE OF VARIABLES) Given an X-WSBPEL \((\text{CFG}_P, \text{XPATH})\), a q-use (or \(\text{use}_v\) for short) of a variable \(v\) is either (i) an occurrence of \(v\) at node \(n\) of \(\text{CFG}_P\) 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\(\text{XPATH}\).

Similar to variable definitions, a variable usage that does not satisfy Definition 4 is named as \(\text{use}_v\). We denote the set of all \(\text{use}_v\) in an X-WSBPEL by \(\text{Use}_v\). In Figure 7, the conceptual variable \(x\) at \(R_3\), as used by "\(y\mid (x,y)\in\text{EDGES}^*(y)\)" is also a \(\text{use}_v\). 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 \(\alpha\) for a variable \(v\) is a triple \((v, n_\alpha, n_\beta)\) such that \(v\) is a q-def at \(n_\alpha\) and a q-use at \(n_\beta\) and there is a definition-clear sub-path (using inorder traversal) with respect to \(v\) from \(n_\alpha\) to \(n_\beta\).

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 (CFG\( _R \), XPath) if and only if, for each XRG-executed 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 \( XQ(\text{UserAddress}, //\text{city}) \) once may not define the variable \( w \) at \( R_6 \) because, when \( V \) at \( R_6 \) 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 (CFG\( _R \), XPath) if and only if, for each query-du association \( \alpha \), there is at least one test case \( t \in T \) such that \( \text{def\_clear}(\alpha) \) 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 (CFG\( _R \), XPath) if and only if, for each predicate query-def-use association \( \alpha \), 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

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Applications</th>
<th>Element</th>
<th>LOC</th>
<th>Query</th>
<th>Query-pu</th>
<th>Query-du</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ATM [4]</td>
<td>94</td>
<td>180</td>
<td>3</td>
<td>12</td>
<td>35</td>
</tr>
<tr>
<td>C</td>
<td>DSLService [27]</td>
<td>50</td>
<td>123</td>
<td>3</td>
<td>11</td>
<td>47</td>
</tr>
<tr>
<td>D</td>
<td>GYMLocker [4]</td>
<td>23</td>
<td>52</td>
<td>2</td>
<td>9</td>
<td>23</td>
</tr>
<tr>
<td>E</td>
<td>LoanApproval [4]</td>
<td>41</td>
<td>102</td>
<td>2</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>G</td>
<td>Purchase [4]</td>
<td>41</td>
<td>125</td>
<td>2</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>H</td>
<td>TripHandling [4]</td>
<td>94</td>
<td>170</td>
<td>6</td>
<td>14</td>
<td>33</td>
</tr>
</tbody>
</table>

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-query-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

<table>
<thead>
<tr>
<th>Category</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-BPEL</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>in-XPath</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>in-WSDL</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

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 test criteria by categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Fault-detecting rates</th>
<th>Box-whisker usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Criterion</td>
<td>Min.</td>
</tr>
<tr>
<td>in-BPEL</td>
<td>Random</td>
<td>0.286</td>
</tr>
<tr>
<td></td>
<td>All-queries</td>
<td>0.524</td>
</tr>
<tr>
<td></td>
<td>All-query-pu</td>
<td>0.524</td>
</tr>
<tr>
<td></td>
<td>All-query-du</td>
<td>0.667</td>
</tr>
<tr>
<td>in-XPath</td>
<td>Random</td>
<td>0.095</td>
</tr>
<tr>
<td></td>
<td>All-queries</td>
<td>0.524</td>
</tr>
<tr>
<td></td>
<td>All-query-pu</td>
<td>0.525</td>
</tr>
<tr>
<td></td>
<td>All-query-du</td>
<td>0.571</td>
</tr>
<tr>
<td>in-WSDL</td>
<td>Random</td>
<td>0.167</td>
</tr>
<tr>
<td></td>
<td>All-queries</td>
<td>0.550</td>
</tr>
<tr>
<td></td>
<td>All-query-pu</td>
<td>0.556</td>
</tr>
<tr>
<td></td>
<td>All-query-du</td>
<td>0.778</td>
</tr>
<tr>
<td>Overall</td>
<td>Random</td>
<td>0.183</td>
</tr>
<tr>
<td></td>
<td>All-queries</td>
<td>0.533</td>
</tr>
<tr>
<td></td>
<td>All-query-pu</td>
<td>0.533</td>
</tr>
<tr>
<td></td>
<td>All-query-du</td>
<td>0.667</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Subject</th>
<th>Random / all-queries</th>
<th>Random / all-query-pu</th>
<th>Random / all-query-du</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atm</td>
<td>2.16</td>
<td>2.13</td>
<td>2.15</td>
</tr>
<tr>
<td>BuyBook</td>
<td>9.64</td>
<td>9.66</td>
<td>9.64</td>
</tr>
<tr>
<td>DSLService</td>
<td>3.61</td>
<td>3.59</td>
<td>2.78</td>
</tr>
<tr>
<td>GYMLocker</td>
<td>2.58</td>
<td>2.57</td>
<td>2.18</td>
</tr>
<tr>
<td>LoanApproval</td>
<td>8.13</td>
<td>8.01</td>
<td>7.97</td>
</tr>
<tr>
<td>Marketplace</td>
<td>13.40</td>
<td>13.13</td>
<td>12.06</td>
</tr>
<tr>
<td>Purchase</td>
<td>3.31</td>
<td>3.18</td>
<td>2.85</td>
</tr>
<tr>
<td>TripHanding</td>
<td>3.47</td>
<td>3.40</td>
<td>2.48</td>
</tr>
<tr>
<td>Average</td>
<td>5.13</td>
<td>5.05</td>
<td>4.29</td>
</tr>
</tbody>
</table>

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.

Garcia-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


