Automated Testing of WS-BPEL Service Compositions: A Scenario-Oriented Approach

Chang-ai Sun, *Member, IEEE*, Yan Zhao, Lin Pan, Huai Liu, *Member, IEEE*, and Tsong Yueh Chen, *Member, IEEE*

Abstract—Nowadays, Service Oriented Architecture (SOA) has become one mainstream paradigm for developing distributed applications. As the basic unit in SOA, Web services can be composed to construct complex applications. The quality of Web services and their compositions is critical to the success of SOA applications. Testing, as a major quality assurance technique, is confronted with new challenges in the context of service compositions. In this paper, we propose a scenario-oriented testing approach that can automatically generate test cases for service compositions. Our approach is particularly focused on the service compositions specified by Business Process Execution Language for Web Services (WS-BPEL), a widely recognized executable service composition language. In the approach, a WS-BPEL service composition is first abstracted into a graph model; test scenarios are then derived from the model; finally, test cases are generated according to different scenarios. We also developed a prototype tool implementing the proposed approach, and an empirical study was conducted to demonstrate the applicability and effectiveness of our approach. The experimental results show that the automatic scenario-oriented testing approach is effective in detecting many types of faults seeded in the service compositions.

Index Terms—Service Oriented Architecture, service compositions, Business Process Execution Language for Web Services, scenario-oriented testing.

1 Introduction

Service Oriented Architecture (SOA) [22] has been widely applied into the development of various distributed applications. Web services, the basic applications in SOA, are often developed and owned by a third party, and are published and deployed in an open and dynamic environment. A single Web service normally provides limited functionalities, so multiple Web services are expected to be composed to implement complex and flexible business processes. Business Process Execution Language for Web Services (WS-BPEL) [14] is a popular language for service compositions. In the context of WS-BPEL, all communications among Web services are via standard eXtensible Markup Language (XML) messages [35], which provides a perfect solution to address the challenging issues in a distributed and heterogeneous environment, such as data exchange and application interoperability. Moreover, Web services inside service compositions can be easily replaced to cater for the quickly changing business requirements and environments due to its loosely coupled feature. However, ensuring the quality of such loosely coupled service compositions becomes difficult yet important.

Testing is a practical and feasible approach to the quality assurance of service-based systems [4], [27]. It mainly involves two aspects, namely testing individual Web services and testing service compositions. The former is usually done by service developers, and lots of testing techniques are available [21], [29], [25], while the latter is left for service consumers, which corresponds to the integrated testing. However, service composition testing is greatly different from the traditional integrated testing in two aspects. First, Web services are developed and tested independently. Thus, it is hard for service developers to expect all possible scenarios. Second, services within compositions are usually abstract ones and only bound to concrete Web services at run-time. This run-time binding delays the execution of testing and calls for the on-the-fly testing techniques. In this context, traditional integrated testing techniques are not applicable.

In the previous work [26], [30], [39], a model-based approach has been proposed to automatically generating a set of test scenarios based on UML activity diagram specifications. With this approach, testers can test on demand the corresponding programs whose behaviors are described by UML activity diagrams. WS-BPEL specifications in nature are very similar to UML activity diagram specifications because they are both based on the state machine and provide the mechanism for supporting concurrent control flows. Unlike UML activity diagram specifications, WS-BPEL specifications are executable programs rather than workflow models.

In this paper, based on our recent study [28], we propose an automatic scenario-oriented testing approach for WS-BPEL service compositions. This approach leverages the previous work [26], [30], [39] on testing UML activity di-

A preliminary version of this paper appeared at the 12th International Conference on Quality Software (QSIC 2012) [28].

C.-A. Sun, Y. Zhao, and L. Pan are with School of Computer and Communication Engineering, University of Science and Technology Beijing, Haidian 100083 Beijing, China (e-mail: casun@ustb.edu.cn).

H. Liu is with Australia-India Research Centre for Automation Software Engineering, RMIT University, Melbourne 3001 VIC, Australia (email: huai.liu@rmit.edu.au).

T. Y. Chen is with Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn 3122 VIC, Australia (email: tychen@swin.edu.au).

All correspondence should be addressed to both C.-A. Sun and H. Liu.

agrams in the context of SOA, and addresses the challenges of testing loosely coupled and run-time binding service compositions. In our approach, an abstract test model is first defined to represent WS-BPEL processes. Then, test scenarios are generated from the test model with respect to the given coverage criteria. Finally, test data are generated and selected to drive the execution of test scenarios. However, in the preliminary study, the approach, especially the final step of test case generation, was not fully automated. In this paper, we not only present basic ideas of the approach, but also propose novel techniques for fully automating the proposed approach. We also developed a prototype tool that implements the proposed approach in the SOA environment. In addition, an empirical study has been conducted to demonstrate our approach and validate its applicability and effectiveness.

The main contributions of this paper, together with its preliminary version [28], are threefold:

(i) We propose a scenario-oriented testing framework for WS-BPEL service compositions, including a graph model for WS-BPEL specifications, a set of mapping rules and an algorithm for converting WS-BPEL specifications to the graph model, an algorithm for generating test scenarios with respect to a specific criterion, and a constraint-based technique of test data generation;
(ii) We develop a tool to automate the proposed scenario-oriented testing framework and algorithms; and
(iii) We validate the feasibility of the proposed approach and evaluate its fault-detection effectiveness via an empirical study.

The rest of the paper is organized as follows. Section 2 introduces underlying concepts related to WS-BPEL, scenario-oriented testing, and constraint solving techniques. Section 3 proposes the scenario-oriented testing approach for WS-BPEL service compositions. Section 4 describes the prototype tool developed by us. Section 5 describes an empirical study where the proposed approach is used to test two real-life WS-BPEL service compositions. The experimental results are discussed in Section 6. Section 7 discusses related work. Section 8 concludes the paper and proposes the future work.

2 Background

In this section, we introduce underlying concepts and techniques of the proposed approach.

2.1 Business Process Execution Language for Web Services (WS-BPEL)

For a service-based system, a bundle of Web services need to be coordinated and each service is expected to execute the predefined functionalities. For such coordination, there are two representative ways, namely orchestration and choreography [22]. Among them, orchestration is more widely recognized and adopted in practice. WS-BPEL is an executable service composition language which specifies business processes by orchestrating Web services, and exports the process as a composite Web service described by the Web Service Description Language (WSDL [34]). In this way, individual WS-BPEL process can be used as a basic service unit to participate in the more complex business processes.

A WS-BPEL process often consists of four sections, namely partner link statements, variable statements, handler statements, and interaction steps. Activities are basic interaction units of WS-BPEL processes, and are further divided into *basic* activities and *structural* activities. Basic activities execute an atomic execution step, including *assign*, *invoke*, *receive*, *reply*, *throw*, *wait*, *empty*, and so on. Structural activities are composites of basic activities and/or structural activities, including *sequence*, *switch*, *while*, *flow*, *pick*, and so on.

WS-BPEL has standard control structures, such as *sequence*, *switch*, and *while*. In addition, WS-BPEL provides *concurrency* among activities via flow activities and *synchronization* via link tags within flows. Each *link* has a *source* activity and a *target* activity. A transition condition, which is an XPath Boolean expression, can be associated with some links. A transition can happen only when the associated conditions are satisfied, that is, executing the target activity after completion of the source activity. If transition conditions are not explicitly specified, their default values are true, indicating that target activity will always be performed after executing the source activity. If an activity is the target activity of multiple links, the activity should have an associated join condition, and only when all the incoming links are defined and its join condition is true, can the activity be enabled. Otherwise, if the join condition is false, the activity is not executed and the effect is propagated downstream to the subsequent activities. Through the flow and link structure, WS-BPEL provides a "multiple-choice" style workflow pattern [32], and multiple outgoing flows may be enabled simultaneously representing concurrent behaviors.

2.2 Scenario-oriented testing

From the workflow point of view, the functionalities of a system can be described as a set of scenarios. A scenario usually represents an execution path of a software system. In this sense, one needs to first derive a set of possible test scenarios and then execute these scenarios with test data and observe their behaviors. Test data associated with a specific test scenario can be derived by analyzing and solving the conditions along the associated path. If an execution of scenarios does not happen as expected, then a fault is detected.

WS-BPEL provides the mechanism to specify concurrent behaviors in workflows. This feature gives rise to new challenges when testing those service compositions with concurrency behavior. First, concurrency behaviors are nondeterministic and thus difficult to be repeated. Secondly, the concurrency mechanism introduces nonstructural elements, which are more difficult to be tested than common control flows. Finally, comprehensive testing of concurrent behaviors significantly increases the number of possible test scenarios.

To decide to what extent concurrent elements should be tested, Sun [26] has proposed three coverage criteria as follows:

- Weak concurrency coverage. Test scenarios are derived to cover only one feasible sequence of parallel processes, without considering the interleaving of activities between parallel processes.
- Moderate concurrency coverage. Test scenarios are derived to cover all the feasible sequences of parallel processes, without considering the interleaving of activities between parallel processes.
- Strong concurrency coverage. Test scenarios are derived to cover all feasible sequences of activities and parallel processes.

Although these test coverage criteria were first proposed for concurrency testing in general, they are applicable to testing concurrent behaviors in WS-BPEL programs. All these test coverage criteria require covering each activity in parallel at least once. Among them, weak coverage and moderate coverage criteria require the parallel activities to be tested in a sequential way, while strong coverage criterion which considers all possible combinations of activities and transitions is therefore impractical due to its huge costs.

2.3 Constraint solving techniques

Constraint solving techniques can enable applications such as extended static checking, predicate abstraction, test case generation and bounded model checking [7]. Especially, constraint solving plays an important role in test case generation for the purpose of coverage or a particular type of property verification, such as bug finding and vulnerability detection [41]. Furthermore, constraint solver-based testing tools enable more precise analysis with the ability to generate bug-revealing inputs.

For the past decade, researchers developed a variety of constraint solvers [18]. Among them, Z3 [7] is one of the most powerful constraint solvers with features of supporting linear real and integer arithmetic, fixed-size bit-vector, uninterpreted functions, extensional arrays, quantifiers, multiple input formats, and extensive APIs (including C/C++/.NET/Java/Python APIs). Z3-str [41] is an extension of Z3 supporting combined logics over strings and non-string operations. It supports three sorts of logic: (i) string-sorted terms include string constants and variables of arbitrary length; (ii) integer-sorted terms are standard, with the exception of the length function over string terms; (iii) Boolean operators are used to combine atomic formulas which are equations over string terms and equalities or inequalities over integer terms. In this paper, we leverage advances in constraint solvers and use Z3 and Z3-str as constraint solver to generate the relevant input values of a path condition, which is to be discussed later.

3 Scenario-Oriented Testing for WS-BPEL Service Compositions

The proposed scenario-oriented testing approach for WS-BPEL service compositions is sketched in Figure 1. In the context of WS-BPEL specifications, a test scenario corresponds to a set of activities and transitions. The number of test scenarios may be very huge when service compositions are complex. One key issue is how to generate a set of test scenarios according to a certain coverage criteria.

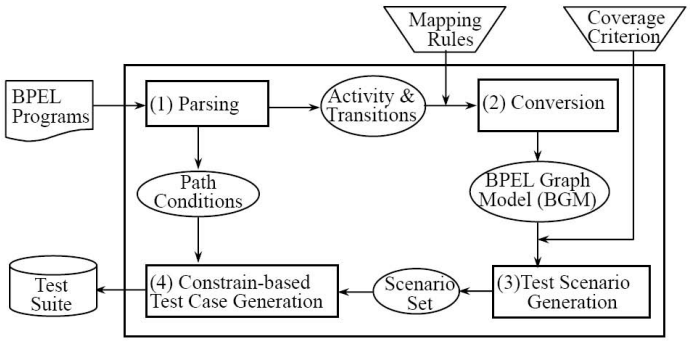


Fig. 1. Overview of our approach

Furthermore, WS-BPEL service compositions may be subject to frequent changes in order to cater for quickly changed business requirements and dynamic environments. It may be very tedious and difficult to generate test scenarios manually, especially for large and complex WS-BPEL compositions. Therefore, the scenario-oriented testing for service compositions should be automated as much as possible. In this section, we elaborate the main steps of our approach and show how the above mentioned issues are addressed when applying the approach to WS-BPEL programs[1].

3.1 WS-BPEL Graph Model (BGM)

WS-BPEL programs are represented as XML files, which are greatly different in syntax from those programs written in traditional programming languages, such as C or Java. On the other hand, WS-BPEL programs are well structured and hence can be easily analyzed by means of available XML analysis techniques, such as Document Object Model (DOM) [33] and Simple API for XML (SAX) [24]. When we generate test scenarios from WS-BPEL programs, only those elements defined in the interaction section make sense, and thus we can skip over those elements defined in partner links variables and handler sections. Furthermore, we can also skip over those attributes of an activity that make no contributions to the construction of test scenarios.

To make the testing task simple and effective, we first define an abstract test model called WS-BPEL Graph Model (BGM), which only considers activities and their relationships within WS-BPEL specifications. The objective of BGM is two-fold. First, one can use it to convert a complex WS-BPEL program into a simple graph, which is formal and easy for analysis. Second, it abstracts activities with the similar semantics as a type of node, thus reducing the quantity of control structure types. For example, both the *switch* and *pick* types are used to specify the optional control logic, and they should share the same notations in BGM.

The BGM of a WS-BPEL program is an extended graph $BGM =< Nodes, Edges >$. A node in $Nodes$ corresponds to a WS-BPEL activity and is represented as an entry $< id, responseid, outing, type, name >$, where

- id is the unique identification of an activity. It starts from zero and each activity inside optional or parallel activities are labeled in a depth-first way.

1. In this paper, we use WS-BPEL compositions and WS-BPEL programs interchangeably.

TABLE 1
Definitions of node types in BGM

Node type	Description
initial	The *beginning* of interactions
end	The *end* of interactions
action	A *normal* activity
branch	The beginning of an *optional* activity
merge	The end of an *optional* activity
fork	The beginning of *parallel* activities
join	The end of *parallel* activities
cycle	A *loop* activity

- $outing$ refers to the number of subsequent activities of the current activity.
- $type$ refers to the type of the current activity. Node types in BGM are summarized in Table 1.
- $name$ refers to the name of the current activity.
- $responseid$ is used to identify the hierarchy of WS-BPEL activities. The definitions of different node type's $responseid$ are further explained as follows.
  - $responseid$ of an $action$ or $initial$ node denotes the next non-normal node;
  - $responseid$ of a $branch$ node denotes its matching $merge$ node;
  - $responseid$ of a $fork$ node denotes its matching $join$ node;
  - $responseid$ of a $merge$ node or $join$ node denotes its next node after the $merge$ or $join$ node, respectively;
  - $responseid$ of an end node denotes itself;
  - $responseid$ of a $cycle$ node denotes itself.

An edge in $Edges$ corresponds to a transition in WS-BPEL and is represented as an entry $< iID, oID >$, where

- iID refers to the ID of the incoming activity of the transition;
- oID refers to the ID of the outgoing activity of the transition.

3.2 Conversion of WS-BPEL program into BGM

We first classify activities of WS-BPEL specifications into five types. Among them, normal activities are the basic activity, while others are structural activities.

- $Normal$ activities refer to an atomic execution unit.
- $Sequential$ activities refer to that their child activities are executed in the sequential order, such as the $sequence$ and $scope$ activity.
- $Optional$ activities refer to that only one branch of their child activities can be executed, such as the $switch$, $if/else\ if/else$, $pick$ activity.
- $Loop$ activities refer to that their child activities are executed all the time until some conditions are satisfied.
- $Parallel$ activities refer to that their child activities are executed simultaneously, such as the $flow$ activity.

In order to convert WS-BPEL specifications into BGMs, we define a set of mapping rules with respect to each type of WS-BPEL activities, as follows.

- For the $normal$ activities, they are directly mapped to action nodes.
- For the $sequential$ activities, a sequence of nodes are created and each node corresponds to a child activity; meanwhile, an edge is added between each pair of nodes.
- For the $optional$ activities, they are mapped to $branch$-$merge$ node pairs, and each branch is mapped to a child node.
- For the $loop$ activities, their child activities are mapped to a sequence of nodes, and a $cycle$ node is added in the end whose outgoing edges are towards the first node in the loop and the first node after the loop.
- For the $parallel$ activities, they are used to support concurrency with flows in WS-BPEL. For those activities without source or target activities, they are executed in parallel. In this context, the $< flow >$-$< /flow >$ pairs are mapped to $fork$-$join$ node pairs, and each parallel branch is mapped to a child node of $fork$ and the father node of $join$. For those child activities that have the source and target elements within the flows, transitions are enabled depending on whether the link's conditions are satisfied.

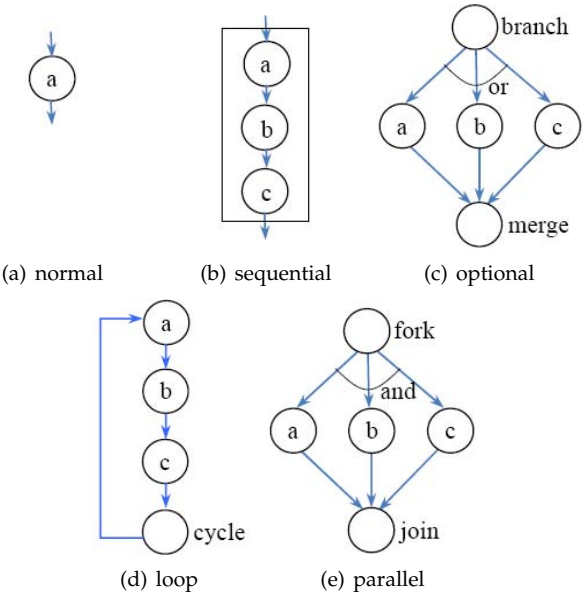


Fig. 2. Illustration of mapping rules

Figure 2 illustrates these mapping rules. Note that for the structural activities ($sequential$, $optional$, $loop$, and $parallel$), their child activities can be either basic or structural activities. The mapping rules discussed above can be applied recursively.

Based on these mapping rules, we propose a recursive conversion algorithm (Algorithm 1), which can be used to automatically convert a WS-BPEL program into a BGM. The proposed algorithm first gets the number of top activities of the current WS-BPEL program, and then converts each activity according to its type. The conversions are conducted following the above-mentioned mapping rules.

The algorithm is able to convert complex and nested structural activities. Its time complexity is proportional to

Algorithm 1 $Covert(B, G)$ converting a WS-BPEL program (B) to a WS-BPEL graph model (G)

1: $n \leftarrow getTopActivityNumber(B)$
2: **for all** $i = 1, 2, \cdots, n$ **do**
3: currentActivity $\leftarrow getActivity(B, i)$
4: **if** currentActivity is a *normal* activity **then**
5: Create an *action* node a and add a to G
6: **else if** currentActivity is a *sequential* activity **then**
7: $BS \leftarrow getBPELSegment$(currentActivity)
8: $Convert(BS, G)$
9: **else if** currentActivity is a *branch* activity **then**
10: Create a *branch* node b and add b to G
11: $BS \leftarrow getBPELSegment$(currentActivity)
12: $Convert(BS, G)$
13: Create a *merge* node m and add m to G
14: **else if** currentActivity is a *loop* activity **then**
15: $BS \leftarrow getBPELSegment$(currentActivity)
16: $Convert(BS, G)$
17: Create a *cycle* node c and add c to G
18: **else if** currentActivity is a *parallel* activity **then**
19: Create a *fork* node f and add f to G
20: $BS \leftarrow getBPELSegment$(currentActivity)
21: $Convert(BS, G)$
22: Create a *join* node j and add j to G
23: **end if**
24: **end for**

the number of basic activities, that is, $O(n)$ where n denotes the number of basic activities.

As an illustration, we apply the above mapping rules to SupplyChain described in Section 5.2. The resulting BGM is shown in Figure 3.

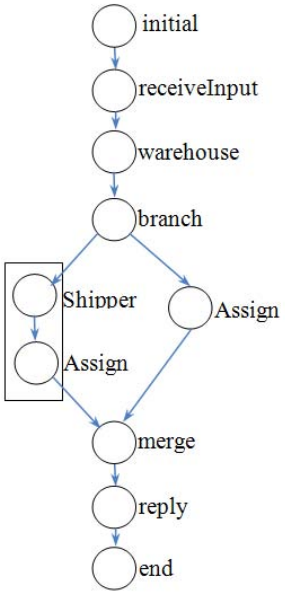


Fig. 3. The resulting BGM for the SupplyChain service composition

3.3 Generation of test scenarios based on BGM

The abstract BGM provides a formal test model on the basis of which we can easily define algorithms to automatically generate test scenarios with respect to the given coverage criterion. As an illustration, we proposed Algorithm 2 to generate test scenarios from BGM with respect to the weak concurrency coverage discussed in Section 2.2.

Algorithm 2 $WeakCoverage$(Node $start$, Node end) generating test scenarios from BGM (G) with respect to weak concurrency coverage

1: **if** $start \neq end$ **then**
2: **if** $start$.type ="$action$" **then**
3: $tmpPaths \leftarrow$ WeakCoverage($start$.afterNodes.get(0), end)
4: Add $start$ node to each path in $tmpPaths$
5: **return** $tmpPaths$
6: **end if**
7: **if** $start$.type = "$action$" **or** $start$.type = "$action$" **then**
8: $node \leftarrow getResponseNode(start$.responseid)
9: **for all** $i = 1, 2, \cdots, start$.afterNodes. size() **do**
10: $tmp[i] \leftarrow$ WeakCoverage($start$.afterNodes.get(i), $node$)
11: **end for**
12: $tmpPaths \leftarrow$ WeakCoverage($start$.afterNodes.get(0), end)
13: merge $tmp[]$ and $tmpPaths$ into $resultPaths$
14: **return** $resultPaths$
15: **end if**
16: **if** $start$.type ="$cycle$" **then**
17: $startNode \leftarrow$ the first node in the loop
18: $endNode \leftarrow$ the last node in the loop
19: $tmpPaths1 \leftarrow WeakCoverage(startNode, endNode)$
20: $tmpNode \leftarrow$ the first node after the loop
21: $tmpPaths2 \leftarrow$ WeakCoverage($tmpNode$, end)
22: Merge $tmpPaths1$ and $tmpPaths2$ into $resultPaths$
23: **return** $resultPaths$
24: **end if**
25: **else**
26: Add $start$ node to $resultPaths$
27: **return** $resultPaths$
28: **end if**

The algorithm generates test scenarios from a *start* node to an *end* node recursively. In the first round, the *start* node corresponds to the *initial* node and the *end* node corresponds to the *end* node of a BGM. After that, a *start* node and an *end* node form a segment of BGM, and partial test scenarios are generated according to the types of nodes:

- If the *start* node is a *branch* node or a *fork* node, the BGM segment is divided into two parts. One is from the *start* node (namely *branch* or *fork* node) to the corresponding response node (namely *merge* or *join* node) that can be specified by means of its *responseid*; the other is from the first node next to the corresponding response node to the *end* node. The algorithm generates partial test scenario for each part and then combines them to form the complete test scenario.

- If the *start* node is a *cycle* node, the BGM segment is also divided into two parts. One is from the first node in loop to the last node in loop; the other is from the first node after loop to the *end* node. The algorithm generates partial test scenario for each part and combines them to form the complete test scenario.
- If the *start* node is the *end* node, it means that all nodes have been handled, and the *start* node is added to the generated test scenarios.

The proposed algorithm traverses all nodes once, so its time complexity is proportional to the number of nodes, that is, $O(n)$ where n denotes the number of nodes in BGM. When the algorithm is applied to BGM shown in Fig. 3, two test scenarios are generated: (i) 'initial'→'receiveInput'→'warehouse'→'branch'→'Shipper' →'Assign'→'Merge'→'reply'→'end'; and (ii) 'initial'→ 'receiveInput'→'warehouse'→'branch'→'Assign'→'Merge' →'reply'→'end'.

3.4 Constraint-based test case generation

Scenario-oriented testing is actually a path sensitive testing approach that traverses program paths in a depth-first way. For each program path, a set of constraints on the program's inputs is generated. The conjunction of those constraints is called a *path condition*, which is used to avoid infeasible paths. The constraints are of various kinds: constraint satisfaction problems (e.g. "A or B is true"), simplex algorithm (e.g. "$x \leq 5$"), and others.

Typically, there are three methods for generating test data for a specific program path [40], namely (i)*symbolic execution* is a method of analyzing a program to determine what inputs cause each part of a program to execute; this method assumes symbolic values for inputs rather than concrete inputs as normal execution of the program would; (ii) *genetic algorithm* searches the possible inputs by means of heuristic that mimics the process of natural selection, such as inheritance, mutation, selection, and crossover; and (iii) *constraint solving*. Among these methods, *symbolic execution* is a static approach and mainly suitable for solving linear paths, while *genetic algorithm* is a dynamic approach requiring the execution of a program multiple times and hence very time-consuming.

Based on the above observations, we employ *constraint solving* to generate test data for a specific program path, and particularly select Z3 [6] as constrain solver. The process of constraint-based test case generation is illustrated in Figure 4. Firstly, the path conditions are identified for each test scenario. Secondly, test data are generated according to the path conditions via constraint solver. Finally, test suites are constructed based on the test data. The details how to automate each step are given in the following.

- *Identification of path conditions*: To identify the path condition of a specific test scenario, we need to extract constraints that have to be satisfied in order to run the scenario. In the context of WS-BPEL, branch statements (such as "switch", "if", "elseif", and "else") and cycle statements (such as "while", "repeatUntil", and "forEach") are supported; all conditions of branch or cycle nodes on the path are extracted and combined to form an expression of the path conditions. Since WS-BPEL programs are represented as well-structured XML files, the extraction of node's information required in BGM is eased via XML DOM. If the condition part of a branch or cycle node in the test scenario is not empty, it will be added to the path condition.
- *Generation of test data*: As mentioned before, a WS-BPEL program is implemented as a composite Web service. The WS-BPEL program is commonly described by a WSDL file, which clearly declares the signature of operations supported by the Web service. Thus, one can know its input requirements by analyzing WS-BPEL programs. It is tedious and time-consuming to manually generate test data that satisfy the path conditions. In this study, we turn to constraint solver Z3 to automatically generate test data according to the expressions of path conditions. To do that, we first convert the path condition expression into variables that Z3 can accept. During this step, we also simplify the expression to remove redundant or irrelevant variables. Next, we set the value range of each input variables by the analysis of WS-BPEL programs. Finally, Z3 solver is executed to obtain concrete values within the range. Test data are then constructed by composing the concrete values of all variables in the expression.
- *Construction of test suite*: A test case consists of test scenario and its associated test data. Since test data can be generated for each test scenario, a complete test suite can be constructed by generating test data for all test scenarios of the WS-BPEL program.

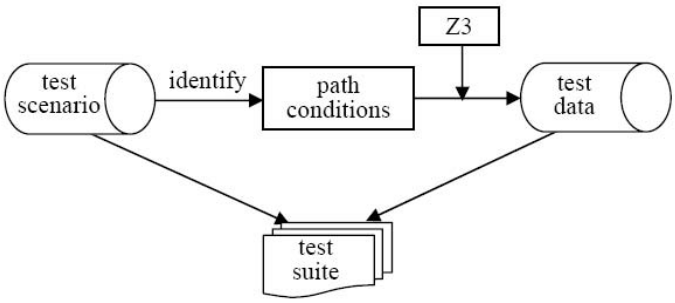


Fig. 4. The process of constraint-based test case generation

4 Prototype Tool

We have developed a prototype tool to automate all the steps of test case generation in the proposed approach. It has the following main features: (i) Automatic generation of a set of test scenarios with respect to a coverage criterion; (ii) Automatic generation of a scenario-oriented test suite whose size is adjustable. Furthermore, it provides an integrated test execution and verification mechanism for WS-BPEL programs: it aids the execution of the WS-BPEL program with generated test cases and the verification of each tests by comparing actual outputs with expected ones.

The tool was developed using Java and its implementation consists of about 3500 lines of codes. Figure 5 depicts the architecture of the tool. The tool consists of three

main components, namely *Test Scenario Generation*, *Test Data Generation*, and *Execution and Verification*. Inside *Test Scenario Generation*, *BPEL Parser* is responsible for parsing WS-BPEL programs; *Converter* converts WS-BPEL programs into BGMs implementing the mapping rules in Section 3.2; *Scenario Generator* implements the algorithms of generating test scenarios with respect to the given concurrency coverage criteria. Inside *Test Data Generation*, *WSDL Parser* is responsible for parsing the WS-BPEL programs; *Path Analyzer* is responsible for constructing of path conditions of test scenarios; *Data Generator* is responsible for generating test data based on path conditions which can drive the execution of the given test scenarios, and its implementation is based on the integration of Z3. Inside *Execution and Verification*, *Proxy* is responsible for executing tests using test suite, and is implemented by integrating ActiveBPEL, a well recognized WS-BPEL engine [1]; *Verifier* is responsible for verifying test results with expected outputs and producing a test report.

Note that the prototype was developed based on the general WS-BPEL standard and thus is not restricted to specific WS-BPEL systems. To equip the prototype with a good adaptability, an interface was reserved to further integrate Apache ODE [2], an open source WS-BPEL engine. More details on the tool is available in [40]. This prototype tool was used in the experiments that evaluate the fault-detection effectiveness of the proposed approach, which is going to be described next.

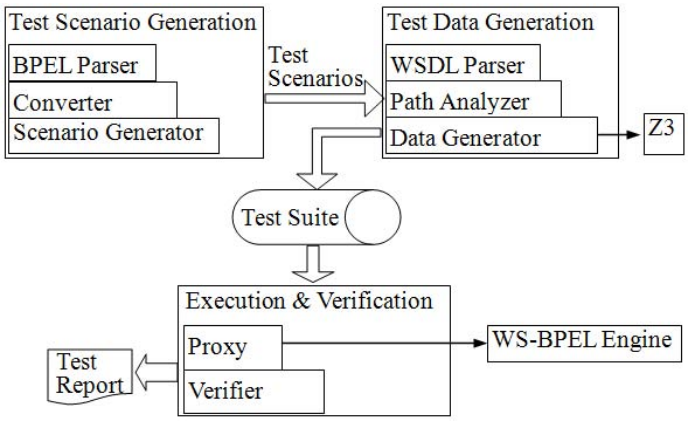


Fig. 5. The architecture of the prototype tool

5 Empirical Study

In this section, we reported on an empirical study where two real-life WS-BPEL programs are used as subject programs, mutation analysis was applied to quantitatively measure the fault-detection effectiveness, and the tool mentioned above was used to aid the experiments.

5.1 Research Questions

In this study, we attempt to answer the following questions.

- RQ1: Is the proposed scenario-oriented approach applicable to WS-BPEL programs?
  In this study, we selected two real-life representative WS-BPEL programs as subject programs, and used the proposed approach and prototype tool to test these two programs.
- RQ2: How effective is the proposed scenario-oriented approach in detecting faults in WS-BPEL programs?
  The fault-detection effectiveness is a major metric for evaluating a testing method. Normally, the more faults a testing method can detect, the more effective it is. In this study, we evaluate the fault-detection effectiveness of the scenario-oriented approach via mutation analysis [8].
- RQ3: Is the proposed approach better than random testing?
  In this study, we compare the fault-detection effectiveness of the proposed approach with that of random testing, which is widely adopted in practice.

5.2 Subject programs

SupplyChain [5] is a WS-BPEL process in the management system of supply chain. Figure 6 illustrates its flowchart labeled with the statement identifiers. It receives three input messages from customers, which are *NameProduct*, *AmountProduct*, and *WarehouseResponse*. Customer sends *Retailer* the booking request, based on which Retailer sends *Warehouse* the supply request. *Warehouse* will response to *Retailer* according to the stock. After receiving the response, *Retailer* will react as follows: If *Warehouse* replies "yes" (that is, the stock is enough), *Retailer* will send shipping request to *Shipper*, who will then confirm by sending "yes" to *Retailer*; if *Warehouse* replies "no" (that is, the stock is not enough), *Retailer* will send the *Customer* the message "Warehouse cannot receive the bill" and cancel the booking. The WS-BPEL service composition for SupplyChain is relatively simple, involving three Web services.

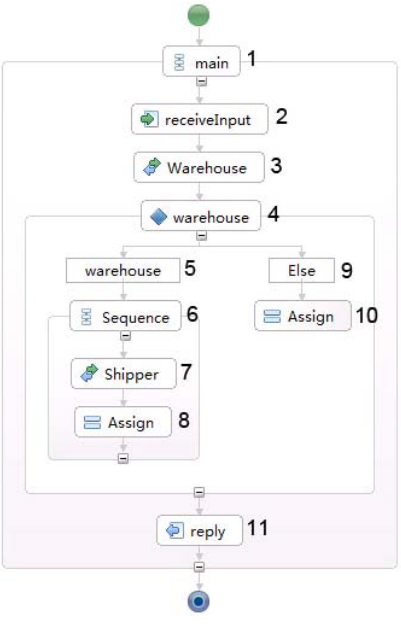


Fig. 6. The BPEL flowchart of the SupplyChain process

SmartShelf [17] receives an input message called commodity, which is composed of three fields, namely *name*, *amount*, and *status*. The process returns an output message which is composed of *quantity*, *location*, and *status*. Figure 7

illustrates its flowchart labeled with the statement identifiers. After receiving the input message, the process compares it with available shelf items and decides whether the amount, location and status of the available items meet the expected requirements. If the amount of available goods on shelf is larger than the amount of commodity, the quantity of message is "Quantity is enough"; otherwise, it transfers goods from warehouse. If the amount of available goods in warehouse is larger than the amount of commodity, the quantity of message is "Quantity is enough"; otherwise, the quantity of message is "Warehouse quantity is not enough". If the name of commodity is not the same as the name of available goods on shelf, it rearranges the goods and returns "Rearrange is done" as the location of message; otherwise it returns "Location is OK". If the status of commodity is larger than the available status of shelf, it sends status to warehouse and returns "Status is fine now" as status of message; otherwise, it returns "Status is ok". The above comparisons are done in parallel. The WS-BPEL service composition for SmartShelf is complex and involves the interactions among 14 Web services. It behaves as a typical concurrent program and hence is very representative.

5.3 Mutant generation

Mutation analysis [8] is widely used to assess the adequacy of a test suite and the effectiveness of testing techniques. It applies some mutation operators to seed various faults into the program under test, in order to generate a set of variants, namely mutants. If a test case causes a mutant to show a behavior different from the program under test, the mutant is said to be "killed".

In this study, we employ mutation analysis to validate the effectiveness of our approach. Though a set of mutation operators was proposed WS-BPEL service compositions [9], [11], [12], only five types of mutation operators are applicable and thus were selected in our experiments: ERR refers to "replacing a relational operator by another of the same type", AIE refers to "removing an else if element or an else element of an activity", ACI refers to "changing the value of the createInstance attribute from 'yes' to 'no"', ASF refers to "replacing a sequence activity by a flow activity", and ASI refers to "exchanging the order of two sequence child activities". In this study, we manually seeded faults into the WS-BPEL programs, because there was no practical tool for this purpose when we started this work. For SupplyChain, we have generated totally 11 mutants; while 26 mutants were generated for SmartShelf. These mutants are summarized in Appendix A. Note that among the 26 mutants for SmartShelf, the mutant #22 is equivalent to the basic program (that is, it always shows the same behavior as the base program), and thus is excluded from the experiment.

5.4 Variables and measures

5.4.1 Independent variables

The independent variables in our experiment are the test case generation techniques. A natural choice for the variable is our scenario-oriented test case generation technique for WS-BPEL service compositions, as described in Section 3. In addition, we used a random testing method as the baseline technique for comparison. In the random testing method, the input parameters are first identified for each program; next, a value range is defined for each input parameter; a concrete value is then randomly generated from the corresponding value range according to the uniform distribution; finally, a test case is constructed by combining the random values.

5.4.2 Dependent variables

We employ two metrics, namely *mutation score* (MS) and *fault discovery rate* (FDR), to measure the effectiveness of our approach. MS is defined as

$$\mathrm{MS}(p, TS) = \frac{N_k}{N_m - N_e}, \quad (1)$$

where p refers to the program under test, TS refers to the test suite used for testing the mutants, N_k refers to the number of mutants killed by TS, N_m refers to the total number of generated mutants, and N_e refers to the number of equivalent mutants. MS intuitively indicates the capability of a test suite killing mutants. The larger the MS is, the more effective a test suite is in killing mutants for the given program.

FDR is defined as

$$\mathrm{FDR}(m, TS) = \frac{N_f}{N_{TS}}, \quad (2)$$

where m refers to a certain mutant, TS refers to the test suite, N_f refers to the number of test cases that can kill m, and N_{TS} refers to the total number of test cases in TS. Intuitively speaking, FDR indicates how effective a test suite is in killing a certain mutant. The larger the FDR is, the more effective a test suite is to kill the given mutant.

5.5 Experiment Environment

The experiments were conducted in the environment of MS Windows 7 with 64-bits, a dual processor of 2.30 GHz, and 2 GB memory. All Web services were implemented in Java language. WS-BPEL programs were developed using Eclipse 4.3.0 and deployed on Apache Tomcat 5.5.33. The prototype tool is based on Z3 4.3.0 for Windows 64 bits and ActiveBPEL 5.0.2.

5.6 Test case generation and data collection

In the experiments, each testing strategy (either our scenario-oriented technique or the random generation method) was used to generate four different test suites for each object program. In the test suites, each test scenario was associated with five, ten, twenty, and fifty test cases. In total, SmartShelf and SupplyChain have 12 and 2 test scenarios, respectively. Therefore, the four test suites for SmartShelf will have contain 60, 120, 240, and 600 test cases, respectively; while the suites for SupplyChain contains 10, 20, 40, and 100 test cases, respectively. In the rest of the paper, we will use TS-60, TS-120, TS-240, TS-600 to represent the test suites for SmartShelf, while TS-10, TS-20, TS-40, TS-100 for SupplyChain.

All test cases were used to test each mutant (except Mutant #22 of SmartShelf, which is an equivalent mutant). The output of the mutant will be compared with that of the

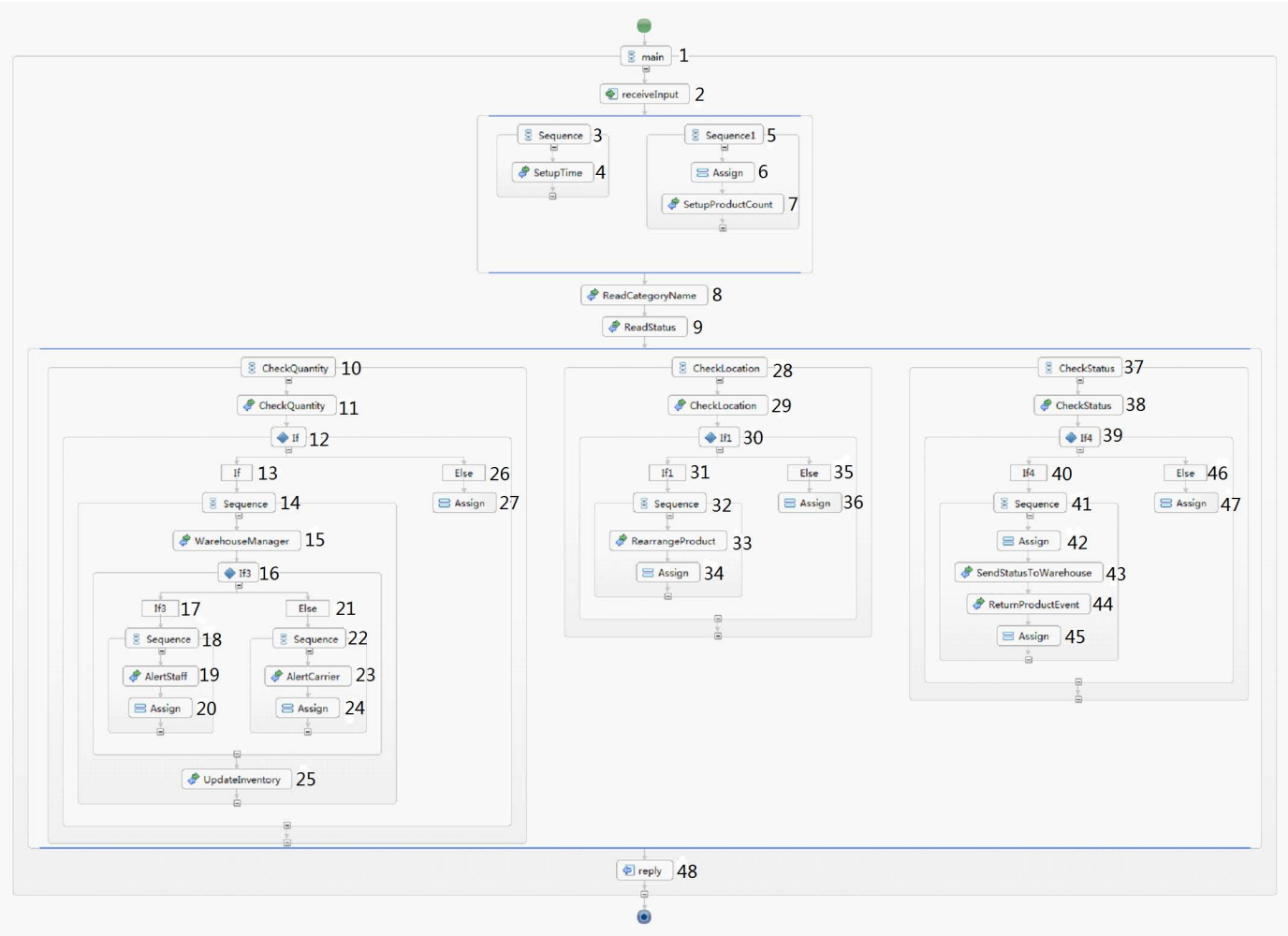


Fig. 7. The BPEL flowchart of the SmartShelf process

base program, which, in this study, was regarded as a test oracle. The testing result ("pass" or "fail") will be recorded to the later calculation of FDR and MS.

5.7 Threats to Validity

The threat to internal validity mainly relates to the implementations of the two testing techniques (our scenario-oriented technique and the random testing method) under this study. The programming work was conducted by several individuals, and was examined by different personnel. We are confident that the testing techniques were correctly implemented.

The threat to external validity is concerned with the selection of object programs. Though only two WS-BPEL service compositions were used in our experiment, they show consistent results (as observed in the next Section 6). Having said that, we cannot say that the similar results will be exhibited on other service compositions.

The threat to construct validity is in the measurements used in this study. Mutation score has been popularly used in the context of mutation analysis for measuring the fault-detection effectiveness of many testing techniques. The other metric, fault discovery rate, is also natural and straightforward to reflect how effective a test suite is in detecting a certain fault.

The most obvious threat to conclusion validity is that we only have a limited number of experimental data. For each subject program, tens of mutants were constructed. For each testing technique, four test suites were generated. Though all the experimental results are consistent, we cannot guarantee that our conclusion is applicable in a more general sense.

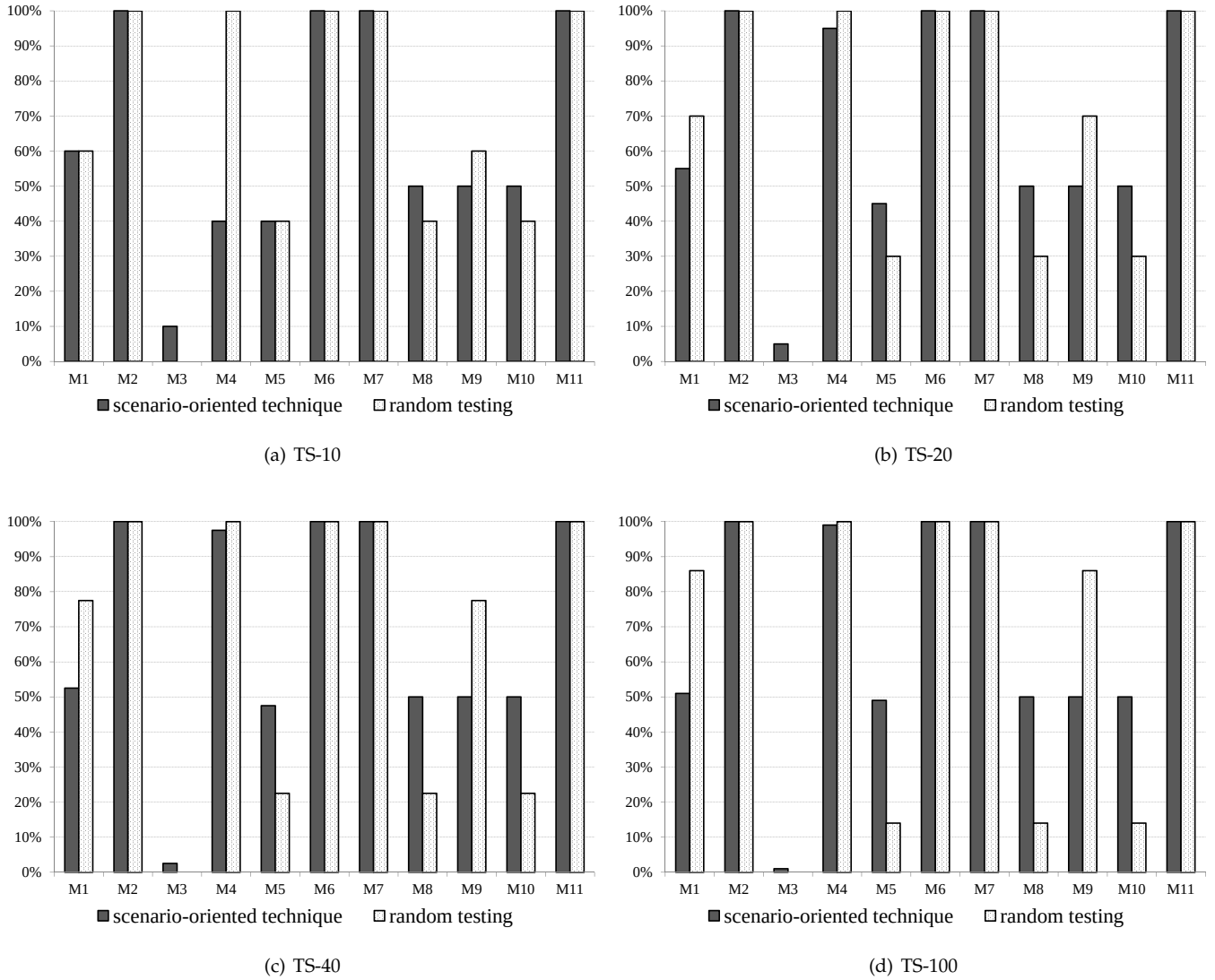
6 Experimental Results

In this section, we report on the experimental results and answer the research questions posed in Section 5.1.

6.1 Results on SupplyChain

The values of FDR on the 11 mutants of SupplyChain are summarized in Figure 8 (Original experimental data are available in Appendix B).

From Figure 8, we can observe that the test suite generated by our approach show different fault-detection effectiveness on different mutants. The effectiveness is also varying with different suite sizes. Such observations are intuitively expected as in the context of software testing;



(a) TS-10

(b) TS-20

(c) TS-40

(d) TS-100

Fig. 8. FDR on SupplyChain

there does not exist a "golden" test suite that is effective in detecting any type of fault. For the fault types ACI, ASF, and ASI, our approach has fairly consistent performance on different suite sizes. However, on the fault types ERR and AIE, we have observed that the effectiveness shows a large variation. In other words, it is relatively uncertain how effective our approach is to detect these two types of faults. Thus, it is recommended that one should pay much attention to ERR and AIE when designing and testing WS-BPEL programs.

In addition, our scenario-oriented technique outperforms random testing for some mutants, while random testing performs better for other mutants. However, we can still observe that FDR of the scenario-oriented technique has lower variation than that of random testing. In other words, our scenario-oriented technique is more reliable than random testing method.

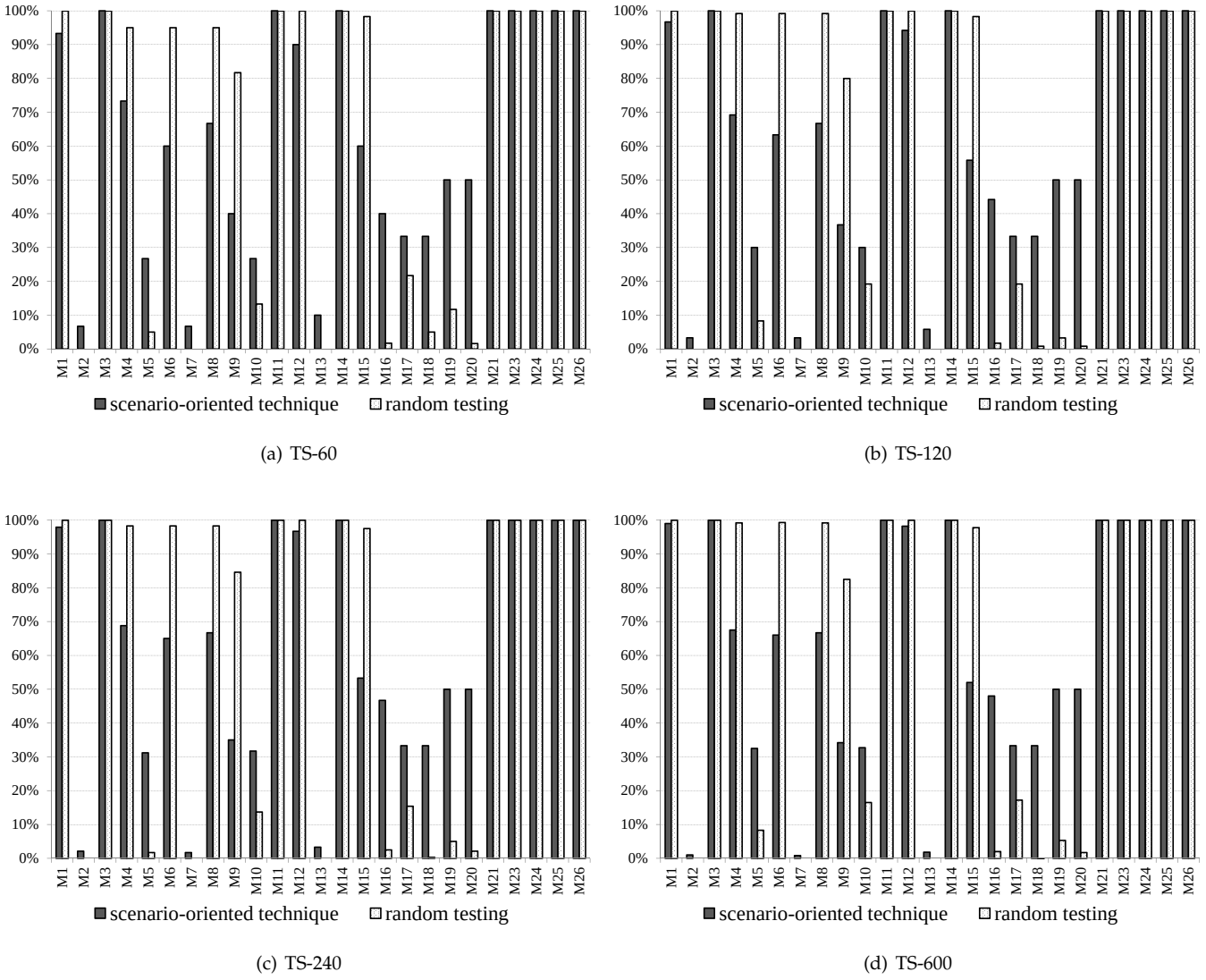
With regard to MS, all four test suites generated by the scenario-oriented technique achieve an MS of 100%, that is, they can kill all the mutants. On contrary, random testing only has an MS of 90.9% (all four random test suites cannot kill the mutant M3). Generally speaking, our scenario-oriented technique is more effective than random testing in detecting various faults.

6.2 Results on SmartShelf

The values of FDR on the 25 mutants of SmartShelf are summarized in Figure 9 (Original experimental data are available in Appendix B).

Our observation based on Figure 9 is similar to those for SupplyChain: The fault-detection effectiveness varies with different mutants, different test suite sizes, and different testing techniques. Our approach delivers consistent effectiveness on the fault types ACI, ASF, and ASI. Nevertheless, the performance of our approach on the fault types ERR and AIE also shows a large variation, which implies that our approach is not steady on these types of faults, and thus much attention should be paid on them in testing. Moreover, the scenario-oriented technique is a more reliable method than random testing.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2466572, IEEE Transactions on Services Computing



(a) TS-60

(b) TS-120

(c) TS-240

(d) TS-600

Fig. 9. FDR on SmartShelf

With regard to MS, its value is always 100% on all four test suite sizes generated by our scenario-oriented technique, while random testing can only achieve an MS of 88% (it could not kill three mutants). It implies that our approach is much more effective than random testing in detecting all seeded faults.

6.3 Answers to Research Questions

Based on the above experimental results and observations, we now answer the research questions:

1) Answer to RQ1 (Applicability): The proposed approach has been successfully employed to test two real-life WS-BPEL programs. Most steps of the proposed approach in the experiments have been automated by the prototype tool. The proposed approach and prototype tool significantly reduce testing efforts, but also make it possible on-the-fly and on demand testing of WS-BPEL programs.
2) Answer to RQ2 (Fault-detection effectiveness): From the experimental results, the proposed scenario-oriented technique shows an MS of 100%, which indicates that generated test suites can detect all seeded faults for both WS-BPEL programs; and the proposed technique also shows a high FDR for most cases, which means that generated test suite is very sensitive to detect faults in WS-BPEL programs. Furthermore, our approach becomes more efficient with the aid of the tool.
3) Answer to RQ3 (Fault-detection effectiveness comparison): The experimental results show that our approach and random testing have a comparable FDR, while the former has a higher mutation score. In addition, the smaller variation of FDR than random testing implies that our approach is a more reliable test technique for WS-BPEL programs.

7 Related Work

WS-BPEL provides a mechanism to build flexible business processes by assembling loosely coupled Web services. Adequately and effectively testing WS-BPEL service composi-

tions is necessary when they are used to execute mission-critical business processes. Some efforts have been reported to test service compositions [3], [23]. We next describe several closely related studies on testing WS-BPEL service compositions.

WS-BPEL service compositions are a kind of concurrent programs with SOA features. To test such concurrent programs, people usually turn to the reachability analysis [31]. Fanjul et al. [15], [16] proposed a model based approach to testing WS-BPEL processes. In their approach, SPIN is employed as a model to generate the test suite for WS-BPEL programs. A transition coverage criterion is employed to select test cases. Since the approach needs to model all possible states of WS-BPEL specifications, it encounters the state space explosion problem when service compositions are complex and thus has limits in practice. Estero-Botaro et al. [10] proposed a test case generation framework for WS-BPEL programs. Their framework is based on a genetic algorithm which generates test suites for mutation testing. To maximize the mutation score of a test suite, the proposed genetic algorithm needs to execute WS-BPEL programs under test many times, which is very time-consuming and thus not efficient. Unlike their approach, our approach uses the adequacy criteria based on the structure of WS-BPEL programs and constraint solving techniques.

Yuan et al. [37] proposed a graph-search based test case generation method for WS-BPEL programs. In their approach, an extended control flow graph (BFG) is defined to represent WS-BPEL programs, and then traverse the BFG model to generate concurrent test paths. Test data are generated using a constraint solving method. A XML schema is employed to represent test cases, which are abstract and thus not executable. No experiments are reported on the effectiveness of their approach. Similarly, Yan et al. [36] proposed an extended control flow graph (XCFG) to represent WS-BPEL programs, and sequential test paths are generated from XCFG. A constraint solver is employed to construct test cases for the generated paths. A case study is reported where 14 sequential paths and 57 combined paths are generated. Only 9 paths are feasible. For the above approaches, experimental results in terms of fault detection capability are missing. Our approach is similar to the two approaches in that both our approach and the above approaches consider test case generation by means of control flows of WS-BPEL programs. Unlike the above approaches, our approach can generate executable test cases on demand and the fault detection capability of our approach is also reported by applying it to two real-life WS-BPEL programs.

Ni et al. [20] recently presented an approach to generating message sequences for testing WS-BPEL programs. Their approach first models the WS-BPEL program under test as a message-sequence graph (MSG), then generates message sequences based on MSG, and finally generates test cases based on MSG. Experiments were performed with six small WS-BPEL programs and the fault-detection effectiveness of their approach was compared with the other techniques. Our approach and their approach explore test case generation for WS-BPEL programs from different directions: The former does this from test scenarios, while the latter from message sequences. On the other hand, both test scenarios and message sequences actually represent a logic path of BPEL programs. Finally, it is not clear to what extent test case generation in their approach can be automated; our approach has been fully automated with the aid of the prototype tool. This is particularly desired for the on-the-fly testing of BPEL programs.

Zhang et al. [38] proposed a model-based approach to generating test cases for WS-BPEL programs. The approach transforms Web service flows into UML 2.0 activity diagrams and then generates test cases from the activity diagram. The approach is very close to our previous work [26], [30], [39], in which we proposed a model-based approach to automatically generating a set of test scenarios for UML activity diagram programs. Unlike the above approaches which generate abstract test cases from a formal or semi-formal representation, our approach presented in this paper generates directly executable test cases and supports different concurrency coverage criteria when generating test scenarios. Furthermore, we developed a prototype tool to automate the proposed approach and conducted case studies on two realistic WS-BPEL programs to show the effectiveness of generated test scenarios and test suite.

8 Conclusion and Future Work

In this paper, we present a scenario-oriented testing approach to address the challenges related to ensuring the quality of WS-BPEL service compositions. This approach first converts WS-BPEL programs into an abstract test model, and test scenarios are then automatically generated from the model with respect to some coverage criteria. For the derived test scenarios, test suites can be obtained automatically. We developed a tool to automate our approach, and an empirical study was conducted to demonstrate how the approach can be applied to real-life service compositions. Experimental results validate the feasibility and effectiveness of the approach.

Using our approach and the prototype, testers can effectively test WS-BPEL service compositions. In the context of SOA, service compositions may face frequent changes, and concrete services are likely to be bound at runtime. Our approach nicely cater for the requirements for testing SOA software in that (1) it supports on demand testing on WS-BPEL service compositions, (2) it saves test efforts significantly through automatic generations of test cases, (3) it can be adapted to the run-time binding because it supports on-the-fly tests. In this sense, our approach does address the challenges caused by the unique features of SOA software.

In our current work, we have achieved the full automation of test case generation, which is a significant improvement from the preliminary study [28]. In the future work, we will investigate how to automate the process of test result verification. Metamorphic testing [19] is a promising approach in this field, and we have conducted some studies on applying this approach into the testing of Web services [29]. It is worthwhile to combine the current work with these previous studies, with the purpose of automating the whole testing procedure for service compositions.

It should be noted that our approach is not restricted to the specific platform. WS-BPEL was designed as a platform-independent language, which make it possible for different

organizations to interconnect their services. Though only two subject programs were used in our empirical study, it does not necessarily imply that our approach is only applicable to them. Having said that, larger-scale empirical studies on different platforms are still required to reinforce the applicability and effectiveness of our approach. In the future work, we will include more different subject programs, and evaluate our approach on the mutants that are automatically (instead of manually) generated by some tools [13]. Moreover, it will be interesting to investigate whether and how our approach can be adjusted and extended to test other types of services (such as big data services) and service compositions (such as mashups). Since the main components of our approach, namely, scenario-oriented testing, Z3 constraint solver, etc., are general techniques rather than specific to WS-BPEL, such adjustment and extension should not be a very difficult task and would be a promising research direction.

ACKNOWLEDGEMENT

We thank Yan Shang from University of Science and Technology Beijing for his help in the experiments reported in this paper, and Yunhui Zheng from Purdue University and Xiao He from University of Science and Technology Beijing for their helpful suggestions on constraint solvers. This research is supported by the National Natural Science Foundation of China under Grant No. 61370061, the Beijing Municipal Training Program for Excellent Talents under Grant No. 2012D009006000002, and the Fundamental Research Funds for the Central Universities under Grant No. FRF-SD-12-015A.

REFERENCES

[1] Active Endpoints. Activebpel engine. http://www.activebpel.org, 2010.
[2] Apache. Apache ODE. http://ode.apache.org/.
[3] M. Bozkurt, M. Harman, and Y. Hassoun. Testing and verification in service-oriented architecture: A survey. *Software Testing, Verification and Reliability*, 23(4):261–313, 2013.
[4] G. Canfora and M. Penta. Service-Oriented Architectures Testing: A Survey. *LNCS 5413, Springer*, pages 78–105, 2008.
[5] M. Chapman, M. Goodner, B. Lund, B. McKee, and R. Rekasius. Sample application supply chain management architecture (version 1.01). Web Services Interoperability Organization, 2003.
[6] CodePlex. Z3. http://z3.codeplex.com/.
[7] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, Lecture Notes in Computer Science Volume 4963, pages 337–340, 2008.
[8] R. A. DeMillo, R. J. Lipton, and F. F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 1(4):31–41, 1978.
[9] J. Dominguez-Jimenez, A. Estero-Botaro, A. Garcia-Dominguez, and I. Medina-Bulo. GAmera: An automatic mutant generation system for WS-BPEL compositions. In *Proceedings of the 7th IEEE European Conference on Web Services*, pages 97–106, 2009.
[10] A. Estero-Botaro, A. Garcia-Dominguez, J. J. Dominguez-Jimenez, F. Palomo-Lozano, and I. Medina-Bulo. A framework for genetic test-case generation for ws-bpel compositions. In *Proceedings of the 26th IFIP WG 6.1 International Conference on Testing Software and Systems*, ICTSS'14, pages 1–16, 2014.
[11] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Mutation operators for WS-BPEL 2.0. In *Proceedings of ICSSEA 2008*, pages 1–7, 2008.
[12] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Quantitative Evaluation of Mutation operators for WS-BPEL compositions. In *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops*, ICST'10, pages 142–150, 2010.
[13] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Dominguez-Jimenez, and A. Garcia-Dominguez. Quality metrics for mutation testing with applications to ws-bpel compositions. *Software Testing, Verification and Reliability*, page in press.
[14] Eviware. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2012.
[15] J. Garcia-Fanjul, C. de la Riva, and J. Tuya. Generation of conformance test suites for compositions of web services using model checking. In *Proceedings of Testing: Academic and Industrial Conference - Practice and Research Techniques*, TAIC PART'06, pages 127–130, 2006.
[16] J. Garcia-Fanjul, J. Tuya, and C. de la Riva. Generating test cases specifications for BPEL compositions of web services using SPIN. In *Proceedings of International Workshop on Web Services - Modeling and Testing*, pages 83–94, 2006.
[17] J. Park, M. Moon, K. Yeom. The BCD view model: business analysis view, service composition view and service design view for service oriented software design and development. In *Proceedings of the 12th IEEE International Workshop on Future Trends of Distributed Computing Systems FTDCS*, pages 37–43, 2008.
[18] S. Kausler and E. Sherman. Evaluation of String Constraint Solvers in the Context of Symbolic Execution. In *Proceedings of 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, pages 259–270, 2014.
[19] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effective are metamorphic relations as a substitute for a test oracle? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.
[20] Y. Ni, S.-S. Hou, L. Zhang, J. Zhu, Z. J. Li, Q. Lan, H. Mei, and J.-S. Sun. Effective Message-Sequence Generation and for Testing BPEL Programs. *IEEE Transactions on Services Computing*, 6(1):7–19, 2013.
[21] J. Offutt and W. Xu. Generating test cases for web services using data pertubation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.
[22] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: A research roadmap. *International Journal on Cooperative Information Systems*, 17(2):223–255, 2008.
[23] H. M. Rusli, M. Puteg, S. Ibrahim, and S. Tabatabaei. A Comparative Evaluation of State-of-the-art Web Service Composition Testing Approaches. In *Proceedings of 6th International Workshop on Automation of Software Test (AST 2011)*, pages 29–35, 2011.
[24] SAX project. Simple API for XML (SAX). http://www.saxproject.org/, 2004.
[25] A. Sharma, T. Hellmann, and F. Maurer. Testing of Web Services - a Systematic Mapping. In *Proceedings of 8th IEEE World Congress on Services (SERVICES 2012)*, pages 346–352, 2012.
[26] C.-A. Sun. A tranformation-based approach to generating scenario-oriented test case from UML activity diagrams for concurrent applications. In *Proceedings of the 32nd Annual IEEE International Computer Sofware and Application Conference*, COMPSAC'08, pages 160–167, 2008.
[27] C.-A. Sun. On open issues on SOA-based software development. *China Sciencepaper Online*, 2011. http://www.paper.edu.cn/index.php/default/releasepaper/content/201107-461.
[28] C.-A. Sun, Y. Shang, Y. Zhao, and T. Y. Chen. Scenario-Oriented Testing for Web Service Compositions Using BPEL. In *Proceedings of the 12th International Conference on Quality Software*, QSIC'12, pages 171–174, 2012.
[29] C.-A. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen. A metamorphic relation-based approach to testing web services without oracles. *International Journal of Web Services Research*, 9(1):51–73, 2012.
[30] C.-A. Sun, B. Zhang, and J. Li. TSGen: A UML activity diagram-based test scenario generation tool. In *Proceedings of the 2009 IEE/IFIP International Symposium on Trusted Computing and Communications*, TrustCom'09, pages 853–858, 2009.
[31] R. Taylor, D. Levine, and C. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.
[32] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Database*.

[33] W3C. Document Object Model (DOM). http://www.w3.org/DOM/, 2005.
[34] W3C. Web Services Description Language Version 2.0. http://www.w3.org/TR/wsdl20/, 2007.
[35] W3C. Extensible Markup Language (XML). http://www.w3.org/XML/, 2008.
[36] 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 the 17th International Symposium on Software Reliability Engineering*, ISSRE'06, pages 75–84, 2006.
[37] Y. Yuan, Z. Li, and W. Sun. A graph-search based approach to BPEL4WS test generation. In *Proceedings of the 2006 International Conference on Software Engineering Advances*, ICSEA'06, page 14, 2006.
[38] G. Zhang, M. Rong, and J. Zhang. A business process of web services testing method based on UML 2.0 activity diagram. In *Proceedings of Intelligent Information Technology Application Workshop*, pages 59–65, 2007.
[39] M. Zhang, C. Liu, and C.-A. Sun. Automated test case generation based on UML activity diagram model. *Journal of Beijing University of Aeronautics and Astronautics*, 27(4):433–437, 2001.
[40] Y. Zhao. A Scenario-based Apporach to Automatic Test Case Generation for BPEL Programs and Its Supporting Tool. Master Thesis, University of Science and Technology Beijing, 2013.
[41] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a Z3-based string solver for web application analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE 2013)*, pages 114–124, 2013.

Chang-ai Sun is a Professor in the School of Computer and Communication Engineering, University of Science and Technology Beijing. Before that, he was an Assistant Professor at Beijing Jiaotong University, China, a postdoctoral fellow at the Swinburne University of Technology, Australia, and a postdoctoral fellow at the University of Groningen, The Netherlands. He received the bachelor's degree in Computer Science from the University of Science and Technology Beijing, China, and the PhD degree in Computer Science from the Beihang University, China. His research interests include software testing, program analysis, and Service-Oriented Computing.

Yan Zhao is a master student in the School of Computer and Communication Engineering, University of Science and Technology Beijing. She received a bachelor degree in Computer Science from the same university. Her current research interests include software testing and Service-Oriented Computing.

Lin Pan is a master student in the School of Computer and Communication Engineering, University of Science and Technology Beijing. He received a bachelor degree in Computer Science from the same university. Her current research interests include software testing and Service-Oriented Computing.

Huai Liu is a Research Fellow at the Australia-India Research Centre for Automation Software Engineering, RMIT University, Australia. He received the BEng in physioelectronic technology and MEng in communications and information systems, both from Nankai University, China, and the PhD degree in software engineering from the Swinburne University of Technology, Australia. His current research interests include software testing, cloud computing, and end-user software engineering.

Tsong Yueh Chen is a Professor of Software Engineering at the Department of Computer Science and Software Engineering in Swinburne University of Technology. He received his PhD in Computer Science from The University of Melbourne, the MSc and DIC from Imperial College of Science and Technology, and BSc and MPhil from The University of Hong Kong. His current research interests include software testing and debugging, software maintenance, and software design.