SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2012; **00**:1–31
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Testing of Data-Centric and Event-Based Dynamic Service Compositions

Waldemar Hummer[1]*, Orna Raz[2], Onn Shehory[2], Philipp Leitner[1], Schahram Dustdar[1]

[1]*Distributed Systems Group, Vienna University of Technology, Austria*
[2]*IBM Haifa Research Labs, Haifa University Campus, Israel*

SUMMARY

This paper addresses integration testing of data-centric and event-based dynamic service compositions. The compositions under test define abstract services which are replaced by concrete candidate services at runtime. Testing all possible instantiations of a composition leads to combinatorial explosion and is often infeasible. We consider data dependencies between services as potential points of failure and introduce the k-node data flow test coverage metric, which helps to significantly reduce the number of test combinations. We formulate a combinatorial optimization problem for generating minimal sets of test cases. Based on this formalization we present a mapping to the model of FoCuS, a coverage analysis tool. FoCuS efficiently computes near-optimal solutions, which are used to automatically generate test instances. The proposed approach is applicable to various composition paradigms. We illustrate the end-to-end practicability based on an integrated scenario which uses two diverse composition techniques: on the one hand, the Web Services Business Process Execution Language (BPEL), and on the other hand WS-Aggregation, a platform for event-based service composition. Copyright © 2012 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Test Coverage, Testing Service-based Systems, Data-Centric Service Compositions, k-Node Data Flow Coverage, Event-based Systems

1. INTRODUCTION

During the last years, the Service-Oriented Architecture (SOA) [1] paradigm has gained considerable importance as a means for creating loosely coupled distributed applications. Services, the main building blocks of SOA, constitute atomic, autonomous computing units with well-defined interfaces, which encapsulate business logic to provide a certain functionality. Today, Web services [2] are the most commonly used implementation technology for service-based applications (SBAs). One of the defining characteristics of services is their composability, i.e., the possibility to combine individual services into service compositions [3]. The de-facto standard for creating Web service compositions is the Business Process Execution Language for Web Services [4] WS-BPEL (or simply BPEL). BPEL uses an XML-based syntax to define the individual activities of the composition and the control flow between them. The mechanism of dynamic binding in SBAs allows to define a required service interface at design time (often called abstract services) and to select a concrete service endpoint from a set of concrete candidate services at runtime.

As any other software system, SBAs require thorough testing, not only of the single participants but particularly of the services in their interplay [5]. Initial testing of a dynamic composite service

*Correspondence to: Distributed Systems Group, Vienna University of Technology, A-1040 Vienna, Austria.
E-Mail: hummer@infosys.tuwien.ac.at

plays a key role for its reliability and performance at runtime. Firstly, the tests may reveal that a concrete service s cannot be integrated, because the results it yields are incompatible with any other service that processes some data produced by s. This incompatibility may be hard to determine with certainty, but testing can indicate potential points of failure. Assume a composition is tested n times, each time with a different combination of concrete services. If the test fails x times ($x < n$), and in all these cases the concrete service s was involved, there is possibly an integration problem with s, which can then be further investigated. Secondly, the test outcome can assist in the service selection process, as it enables operators to favor well-performing service combinations and avoid configurations for which tests failed. Furthermore, upfront testing makes it safer and possibly faster to move to a new binding when a new concrete service becomes available. The practical relevance of integration testing of dynamic service compositions is evidenced by ongoing efforts towards definition of standard service interfaces for various industries. For instance, the TeleManagement Forum (TMF[†]) has released an extensive set of best practices and interfaces for the telecommunications domain, allowing providers to implement standardized business processes and to expose their services to the outside world. The high level of adoption of these standards is illustrated in the TMF's 2009 Case Study Handbook [6] which presents 30 real-world solutions from renowned providers. Similar types of standards and interfaces are used in the airline industry (IATA[‡]), the retail industry (ARTS [§]), and many others.

In practice, testing dynamic composite SBAs is a challenging task for two main reasons. First, traditional white-box testing procedures are only available if the tester has access to the source code or a model of the service processing logic. However, as a general characteristic of service-oriented software engineering, the implementation of services beyond the interface is usually hidden from service users. In the case of Web services, the interface is provided in the form of a WSDL [7] (Web Service Description Language) service description document, combined with XML Schema definitions of the operation inputs and outputs. Second, dynamic service binding is combinatorial in the number of concrete services. That is, for any nontrivial composition, the number of possible combinations may be prohibitively large.

For the first problem, several testing techniques and coverage criteria have been proposed on the service interface level, e.g., [8] or [9]. Moreover, recent studies attempt to bring white-box testing approaches to SBAs by creating *testable services* that reflect and report on the degree of test coverage without revealing the actual service internals [10, 11]. The second problem is considered one of the main challenges in service testing [12], and has previously been addressed in group testing of services [13, 14]. These works present very basic high-level service composition models and propose test case generation for progressive unit and integration testing. Our work builds on their approach, and provides techniques for restricting the combinations of services, as well as for applying the generated tests to concrete Web service composition technology.

This paper focuses on integration testing of dynamic service compositions with an emphasis on data-flow centric coverage goals. We propose coverage criteria which target the inter-invocation data dependencies in dynamic service compositions. We show that such dependencies are important to test because of their potential effect on the correct behavior of the composition. We provide a detailed problem formulation, and present a practical, ready-to-use solution that is both based on existing tooling for software testing and applied to actual Web service technology. The major contributions of our work are threefold:

- We illustrate and formalize a data-centric view of service compositions in a high-level, abstract way. We introduce *k-node data flow* coverage as a novel metric expressing to what extent the data dependencies of a dynamic composition are tested. Based on this test coverage metric, we formulate the problem of finding a minimal number of test cases for a service composition as a combinatorial optimization problem (COP). The outcome of the COP is a set of concrete test instantiations of the compositions, which ensures that all relevant data

[†]TeleManagement Forum; http://www.tmforum.org
[‡]International Air Transport Association; http://www.iata.org
[§]Association for Retail Technology Standards; http://www.nrf-arts.org/

DOI: 10.1002/stvr

dependencies are covered (hard constraints). Limiting the level of desired coverage via the k-node coverage metric helps to significantly reduce the search space of service combinations. Additionally, testers can assign weights to give precedence to services that are known to be less reliable (soft constraints).

- In order to solve the COP, we make use of FoCuS [15], a tool for coverage analysis and combinatorial test design. We provide an automated transformation from the service composition model to the FoCuS data model. The input to FoCuS is constructed from the composition definition (e.g., BPEL) and meta information about the available services. At this point, we further narrow down the search space by specifying past instantiations of the composition, which constitute existing solutions and can be ignored by the solver. The near-optimal solution computed by FoCuS is then transformed back into the service composition model to construct actual test cases.
- We discuss and present the *TeCoS* (Test Coverage for Service-based systems) framework, which is a prototype implementation of the presented approach. TeCoS stores meta information about SBAs, logs invocation traces, and measures different coverage metrics. This combined information is used to generate and execute service composition test cases. We show how different types of SBAs are supported by means of a customizable adapter mechanism, which executes the test cases on target platforms. Our evaluation analyzes various performance characteristics, and demonstrates the end-to-end practicability of the solution.

An introduction to data-centric testing of service-based systems, particularly for compositions defined as BPEL processes, has been given in an earlier version of this paper [16]. In the current paper, we discuss and evaluate the approach in detail, extend the focus to testing of event-based systems, provide an in-depth discussion of how the test results are analyzed to determine incompatible and faulty services, and illustrate the generality and extensibility of the technique by applying it to an additional, fundamentally different, service composition model besides BPEL.

1.1. Approach Synopsis

In this section, we briefly describe the solution from a high level perspective. The details of the comprising concepts and elements are presented in the proceeding sections. Figure 1 depicts the end-to-end view, including all activities, required inputs, and generated output. The input to the first activity is a composition definition document. This activity converts the input document into a corresponding data flow model, which provides an abstraction of the data-related dependencies between invocations of the composition (see Section 3 for details).

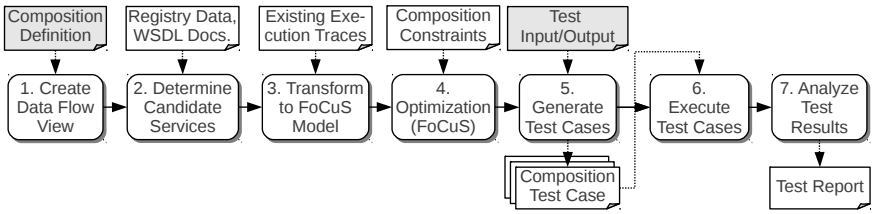

Figure 1. End-to-End Approach

The second activity determines the candidate services for each abstract service identified in the data flow view. Then, in the third activity, the combined information is transformed into a model that complies with the FoCuS API. This activity optionally allows to specify existing execution traces of the composition, which are considered as an existing solution by the solver and hence narrow down the search space. The fourth activity is the optimization process (executed by FoCuS), which computes a near-optimal solution for the model produced by activity three. In addition, the optimizer (FoCuS) receives as input a list of additional composition constraints that specify which services should or should not be used in combination in the generated test cases. In activity five, this solution, together with test input/output combinations, is used to generate the concrete test cases for the composition. These tests are then deployed and executed by activity six. Finally, activity seven

DOI: 10.1002/stvr

analyzes the results, determines incompatibilities and potential points of failure, and summarizes the findings in a test report. Within this process, only the composition definition and the test input/output are defined manually by the composition developer/tester (depicted in gray in the figure). Other documents and *all* of the seven activities are automated and require no human involvement.

1.2. Roadmap

The remainder of this paper is organized as follows. In Section 2 we introduce an illustrative scenario which serves as the basis for discussion. Section 3 introduces a formal model for dynamic data-centric service compositions and defines the coverage goal we aim for. In Section 4, we discuss the details of the combinatorial test design and provide an end-to-end view of the testing approach. Section 5 discusses implementation details and outlines the functionalities of the TeCoS framework. Section 6 contains a thorough evaluation covering various performance and quality aspects, and Section 7 points to related work in the area of testing SBAs and service compositions. Section 8 concludes the paper with an outlook for future work.

2. SCENARIO

We base the description of dynamic data-centric compositions on an illustrative scenario of a tourist who plans a city trip and requests vacant hotel rooms, available flights and visa regulations for the target destination. This functionality is implemented as a composite Web service using BPEL, a business process based composition language, and WS-Aggregation, a distributed platform for event based Web service composition and data aggregation that we have published in earlier work [17, 18, 19]. BPEL is used to model and execute the business process logic of the composition, whereas the WS-Aggregation platform is tailored to event-based processing of Web service data with continuous queries and high data throughput.

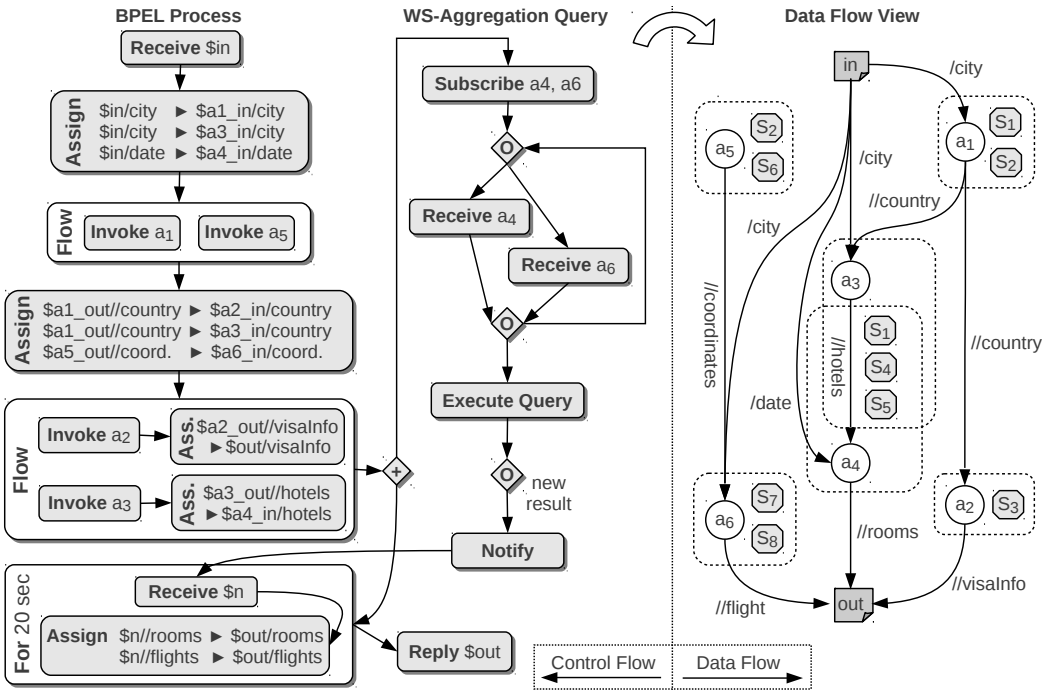

Figure 2. Scenario – Composition Business Logic View (left) and Data Flow View (right)

Figure 2 depicts the corresponding service composition: the left part of the figure contains a graphical representation of the BPEL process and the WS-Aggregation query processing steps (in

DOI: 10.1002/stvr

BPMN notation [20]), and the right part illustrates the data flow between the services of the process. While the BPEL process and WS-Aggregation query are defined by the composition developer, the data flow view can be generated automatically.

The scenario composition receives an input (`in`) and uses six abstract services $\{a_1, ..., a_6\}$ to produce the output (`out`). The input contains two XML elements, `city` and `date`. In BPEL, an abstract service is defined as an `invoke` activity, which is associated with input and output variables, e.g., `$a1_in` and `$a1_out` for service a_1. Between each two abstract services (`invoke` activities) a_x and a_y there is an `assign` activity, which copies the required output of a_x to the input variable of a_y. The respective source and destination are defined via XPath [21] expressions.

Service a_1 determines the country of the given city, a_2 returns the visa information of this country. Services a_3 and a_4 retrieve existing hotels in the specified city/country and available hotel rooms, respectively. Finally, a_5 looks up the user's current geographical coordinates (e.g., using GPS on a mobile device), which are used to find available flights with service a_6. While the BPEL process largely follows a sequential procedure, WS-Aggregation maintains event subscriptions to receive asynchronous notification events from the services a_4 (if new rooms become available) and a_6 (if new flight information is available) and performs continuous queries over these events.

In the left part of Figure 2, arrows indicate the control flow between the activities. The structured `flow` activities in BPEL signify that the contained activities execute in parallel. The `for` activity defines a loop that executes for 20 seconds and collects notifications received from WS-Aggregation. The diamond shapes represent split and merge nodes, denoted as gateways in BPMN. A diamond with a plus sign ("+") represents a parallel gateway (all outgoing paths are executed), and a diamond with a circle is an inclusive gateway (any combination of paths may be taken, zero to all).

The right part of Figure 2 shows the composition business logic view transformed into the composition data flow view. The data flow view used here is a simplified version of the BPEL data dependencies model presented in [22]. Whereas their approach models the actual structure of the BPEL process, our view abstracts from the composition process and is agnostic of control flow instruments, such as the BPEL `flow` or `for` activities. Therefore, our approach lends itself to model data flows in heterogeneous service composition environments (BPEL and WS-Aggregation in our case). The arrows signify the data flow between services. An arrow pointing from a service a_x to a_y, labeled with an XPath expression, means that a part of the output of a_x becomes (part of) the input of a_y. As an additional information, we included the concrete service endpoints in the figure. A dashed rounded box around an abstract service defines the concrete services that can deliver the desired functionality, e.g., the abstract service a_1 is implemented by the concrete Web services s_1 and s_2. At runtime, the composition is instantiated by selecting for each abstract service one of a set of concrete candidate services, often denoted as *dynamic binding* [23]. The service selection process usually happens based on non-functional and QoS [24] (Quality of Service) parameters.

In our scenario we define that the endpoint for a_3 and a_4 must always be the same concrete service (one out of s_1,s_4,s_5). As an example, using s_1 for a_3 and s_4 for a_4 would result in an incompatibility and is not allowed. Note that such constraints are defined as additional information, and are not directly (graphically) reflected in the data flow view.

2.1. Sources of Faults in Dynamic Service Compositions

Even in the case that all concrete candidate services have been tested individually (unit testing), the challenge is to ensure that they also function correctly in their interplay (integration testing). The services in our scenario depend on one another, both directly and transitively, via data flow dependencies. If the services are not properly tested in combination, these dependencies can lead to undesired behavior and defects in the composition. The following list contains an excerpt of potential composition faults which raise the necessity of end-to-end integration testing:

- *Syntactic Data Incompatibility*: The coordinates returned by the geographic service (abstract service a_5 in the composition) are encoded as strings. Assume that the concrete service s_2 uses the notation *51° 28' 38" N*, whereas s_6 uses a dash as delimiter, i.e., *51:28:38:N*. If the concrete flight services s_7 and s_8 (a_6 in the composition) are not able to parse both formats correctly, the combination will likely lead to an exception or a faulty composition result.

DOI: 10.1002/stvr

- *Semantic Data Incompatibility*: Assume that the data formats of s_2 and s_6 are syntactically identical, e.g., both use the format *51:28:38:N* to encode the geographic latitude. Although the reference point most frequently used today is the Greenwich Meridian, in principle the meridian is a matter of convention and historically different meridians have been used (e.g., Paris Meridian). Evidently, if two services do not follow the same semantic data conventions, the composition becomes faulty. Similar problems can occur if the date and time is handled differently by concrete service. For instance, time zones are a key issue in international trade.
- *Security Enforcement Problem*: Security and data privacy are key requirement for service compositions [25, 26]. Assume that the service level agreements (SLAs) between user and composition provider mandate that sensitive data (e.g., location) are not transmitted in plain text. Therefore, service a_5 in the composition uses cryptography to encrypt the coordinates before they are passed to service a_6. If the concrete service receiving the data (s_7 or s_8) has no matching key for decryption, the composition fails and no flight information is received.
- *Network Partitions*: Also the physical deployment of the services needs to be taken into account. Assume that the network is partitioned, i.e., the services are deployed in different domains with local IP addresses that are not accessible across the network. This point is particularly relevant in decentralized compositions where the services interact directly with each other. Moreover, in the centralized execution model of BPEL, it must be ensured that no concrete services are selected which are inaccessible by the composition engine.

2.2. *Runtime Composition Instances*

Dynamic binding is used to select concrete candidate services at composition instantiation time. Table I lists the possible combinations of concrete services for the scenario composition. The column titles contain the composition's abstract services, and each combination has an identifier (#). Each table value represents a concrete service that is used within a certain combination (row) for a certain abstract service (column). In total, 24 combinations exist for the scenario composition.

#	a_1	a_2	a_3	a_4	a_5	a_6	#	a_1	a_2	a_3	a_4	a_5	a_6
c_1	s_1	s_3	s_1	s_1	s_2	s_7	**c_{13}**	**s_1**	**s_3**	**s_1**	**s_1**	**s_2**	**s_8**
c_2	s_2	s_3	s_1	s_1	s_2	s_7	c_{14}	s_2	s_3	s_1	s_1	s_2	s_8
c_3	s_1	s_3	s_4	s_4	s_2	s_7	c_{15}	s_1	s_3	s_4	s_4	s_2	s_8
c_4	**s_2**	**s_3**	**s_4**	**s_4**	**s_2**	**s_7**	c_{16}	s_2	s_3	s_4	s_4	s_2	s_8
c_5	s_1	s_3	s_5	s_5	s_2	s_7	**c_{17}**	**s_1**	**s_3**	**s_5**	**s_5**	**s_2**	**s_8**
c_6	s_2	s_3	s_5	s_5	s_2	s_7	c_{18}	s_2	s_3	s_5	s_5	s_2	s_8
c_7	s_1	s_3	s_1	s_1	s_6	s_7	c_{19}	s_1	s_3	s_1	s_1	s_6	s_8
c_8	s_2	s_3	s_1	s_1	s_6	s_7	**c_{20}**	**s_2**	**s_3**	**s_1**	**s_1**	**s_6**	**s_8**
c_9	s_1	s_3	s_4	s_4	s_6	s_7	**c_{21}**	**s_1**	**s_3**	**s_4**	**s_4**	**s_6**	**s_8**
c_{10}	s_2	s_3	s_4	s_4	s_6	s_7	c_{22}	s_2	s_3	s_4	s_4	s_6	s_8
c_{11}	s_1	s_3	s_5	s_5	s_6	s_7	c_{23}	s_1	s_3	s_5	s_5	s_6	s_8
c_{12}	**s_2**	**s_3**	**s_5**	**s_5**	**s_6**	**s_7**	c_{24}	s_2	s_3	s_5	s_5	s_6	s_8

Table I. Possible Combinations of Services
(The set of compositions printed in bold covers all combinations of service pairs connected by a data flow.)

Services with data dependencies have a direct influence on one another and create a potential point of failure (as outlined in Section 2.1). Hence, for a composition to be tested thoroughly, all concrete service combinations need to be taken into account for service pairs connected by a data flow. Finding the smallest set of test compositions to satisfy this criterion is a hard computational problem (more details are given in Section 4). For our scenario, the size of this set is 6 (the largest combination set results from pairing all services of a_1 with all of a_3), and one possible solution is the set $\{c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}\}$ (bold print in Table I). Although this example can be easily solved optimally, for problem instances with additional real-world constraints (e.g., that two specific services must never be used in combination), it becomes harder to determine the minimum size of the solution set, and finding an optimal solution becomes infeasible for larger instances.

DOI: 10.1002/stvr

3. TESTING OF DYNAMIC DATA-CENTRIC COMPOSITIONS

In this section we establish the key concepts and terminology which build the foundations for testing of dynamic data-centric service compositions. We formalize a unified and platform-independent service composition model in Section 3.1. Based on the composition model we define the test coverage goal that should be achieved in Section 3.2. Section 3.3 illustrates how the platform-independent composition model is mapped to the concrete platforms BPEL and WS-Aggregation.

3.1. Service Composition Model and Composition Test Model

We formally define the service composition model and the composition test model, which serve as the basis for the remaining parts of the paper. Table II describes the elements of the service composition model. The left column contains symbols and variable names, the middle column defines the respective symbol, and the right column provides examples in reference to the scenario.

Symbol	**Description**	**Example**
$A = \{a_1, ..., a_n\}$	Set of abstract services that are used in the composition definition.	$A = \{a_1, ..., a_6\}$
$S = \{s_1, ..., s_m\}$	Set of concrete services available in the service-based system.	$S = \{s_1, ..., s_8\}$
$s : A \rightarrow \mathcal{P}(S)$	Function that returns for an abstract service all concrete services providing the required functionality. $\mathcal{P}(S)$ denotes the power set of S.	$s(a_3) = \{s_1, s_4, s_5\}$
$F \subseteq A \times A$	Set of direct data flows (dependencies) between two services. Possible data flows spanning more than two services can be derived from F (see Section 3.2).	$F = \{(a_1, a_2), (a_1, a_3), (a_3, a_4), (a_5, a_6)\}$
$C \subseteq [A \rightarrow S]$	Domain of all possible runtime composition instances. The composition function $A \rightarrow S$ maps abstract services to concrete services.	$C = \{c_1, c_2, .., c_{24}\}$ (cf. Table I)
$R^+, R^- \subseteq \mathcal{P}((A \rightarrow S) \times (A \rightarrow S))$	Restrictions in the use and combination of concrete services. Service assignments in R^+ *must* always be used in combination: $\forall c \in C, (r_1, r_2) \in R^+ : r_1 \in c \Leftrightarrow r_2 \in c$. Services in R^- *must not* be combined: $\forall c \in C, (r_1, r_2) \in R^- : r_1 \notin c \vee r_2 \notin c$.	$R^+ = \{ (a_3 \mapsto s1, a_4 \mapsto s_1), (a_3 \mapsto s4, a_4 \mapsto s_4), (a_3 \mapsto s5, a_4 \mapsto s_5) \}$
$c \in C$	Concrete runtime composition instance.	c_4: $a_1 \mapsto s_2$, $a_2 \mapsto s_3$, $a_3 \mapsto s_4$, $a_4 \mapsto s_4$, $a_5 \mapsto s_2$, $a_6 \mapsto s_7$

Table II. Service Composition Model

Table III lists the core elements of the composition test model. The set of test composition instances T is a subset of all possible composition instances. The goal we aim for is to keep the size of T small, in order to reduce the effort for test execution. The occurrence count function o indicates the frequency of binding between abstract and concrete services. Finally, we use the function r to map a composition instance to a test result. For simplicity we assume that there are two result states, *success* and *failed*. The mechanism to determine whether a test case has failed or passed is often termed *test oracle* [27]. Details about test oracles in our approach are given in Section 5.5.

Symbol	**Description**	**Example**
$T \subseteq C$	Set of actual test composition instances to be executed. This is the encoding of a solution and the goal is to minimize its size.	$T_{min} = \{c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}\}$

DOI: 10.1002/stvr

$o(s_x, a_y) \stackrel{\text{def}}{=} \lvert\{c \in T : c(a_y) = s_x\}\rvert$	Occurrences of concrete service s_x as implementation of an abstract service a_y in any of the compositions in set T.	$o(s_2, a_5) = 3$ (cf. Table I)
$r : C \to R$, $R = \{success, failed\}$	Function that determines the test result (success, failed) for a given composition instance. This function performs the task of a *test oracle* [27].	$r(c_{20}) = failed$ (assuming that services s_2, s_6 have data incompatibilities)

Table III. Composition Test Model

Additionally, we define the minimum (Equation 1) and maximum (Equation 2) number of uses of any concrete service for a specific abstract service $a_y \in A$ across all compositions in the test set T. The difference between $min(a_y)$ and $max(a_y)$ indicates how uniformly the test cases in T are generated. For instance, if we consider the test set $\{c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}\}$ from Table I, then $min(a_1) = max(a_1) = 3$ because both concrete services s_1 and s_3 are used three times for a_1. However, the concrete services for a_6 are less uniformly distributed: $min(a_6) = 2, max(a_6) = 4$.

$$min(a_y) \stackrel{\text{def}}{=} min(\{o(s_x, a_y) : s_x \in s(a_y)\}), a_y \in A \tag{1}$$

$$max(a_y) \stackrel{\text{def}}{=} max(\{o(s_x, a_y) : s_x \in s(a_y)\}), a_y \in A \tag{2}$$

3.2. *k-Node Data Flow Coverage Metric*

Before proceeding with details of the combinatorial test design, we take a closer look at data dependencies. The data flow view of compositions allows for an analysis of the dependencies between services and possible points of failures. As already mentioned in Section 2.1, services with direct data dependencies are prone to errors. However, indirect dependencies in a sequence, e.g., $a_1 \to a_3 \to a_4$, should be considered as well. In this example, there are two possible dependencies: firstly, if a_3 passes on to a_4 some part of the (unchanged) input it received from a_1, then a hidden, but direct, dependency between a_1 and a_4 is established; secondly, a_3 may operate differently depending on which service is chosen for a_1, and the output of a_3 affects a_4. Hence, we generalize the coverage metrics for dynamic service compositions and introduce the *k-node data flow*.

Definition 1
A **k-node data flow** d_k is a sequence $\langle a^1_{dk}, a^2_{dk}, .., a^k_{dk}\rangle$ of abstract services, $a^1_{dk}, a^2_{dk}, .., a^k_{dk} \in A$, such that $\forall j \in \{1, .., k-1\} : (a^j_{dk}, a^{j+1}_{dk}) \in F$. In the special case where $k = 1$, the list contains only one element: $\langle a^1_d\rangle$. $F_k = \{d^1_k, d^2_k, .., d^l_k\}$ denotes the set of all k-node data flows in a service composition definition.

Definition 2
A **data flow coverage** $cvg(d_k) \in \mathcal{P}(S^k)$ of a k-node data flow d_k denotes the set of possible concrete service assignments along the path of d_k. More precisely, $cvg(d_k)$ is a set of concrete service sequences of length k, such that all possible service assignments with respect to d_k are covered: $\forall s^1_{dk} \in s(a^1_{dk}), ..., s^k_{dk} \in s(a^k_{dk}) : \langle s^1_{dk}, ..., s^k_{dk}\rangle \in cvg(d_k)$. For each of the service combinations $\langle s^1_{dk}, ..., s^k_{dk}\rangle \in cvg(d_k)$ the output of service s^i_{dk} becomes input of service s^{i+1}_{dk}, $i \in \{1, \ldots, k-1\}$.

Definition 3
An SBA is **k-coverage tested** if, for all j-node data flows d_j, $j \in \{1, .., k\}$, there exist test cases such that all service combinations of $cvg(d_j)$ are covered.

Definition 3, stated in other words, asserts that a composition is *k-coverage tested* if all data flow paths of length of at most k have been tested with all possible service combinations. Figure 3 illustrates this for a value of $k = 3$ in our scenario: inter-service dependencies of the data flow view are depicted as trees, and all paths of length 2 and 3 (outlined with light gray background) need to be covered. Note that the k-coverage criterion is only reasonably applicable if the underlying services have been unit tested with appropriate verification and validation techniques. Also, k-node data

DOI: 10.1002/stvr

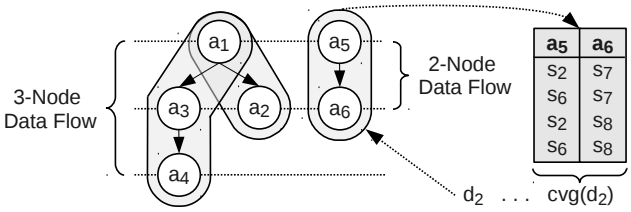

Figure 3. Data Flows in Scenario

flow coverage does not take into account the different content of the data passed between concrete services. Hence, the effectiveness of the metric relies on the input data that is used to execute the test cases. The question of which test input is suitable depends on both the service interfaces (e.g., extreme values) and the problem domain. For systematic methods to generation of appropriate test inputs we refer to previous work, e.g., [28, 29, 9].

3.3. Mapping the Composition Model to Concrete Platforms

To generate executable tests for our service composition, the platform-specific composition model (query model) must be mapped to the platform-independent composition model (as defined in Section 3.1). Reversely, after the test cases have been generated, the concrete composition service bindings must be mapped back to the platform for execution. In the following we discuss the mapping between our composition model and concrete platform models, for BPEL (Section 3.3.1) and WS-Aggregation (Section 3.3.2). Section 5 provides more details about the implementation of the mapping and how the approach is extensible to support further composition paradigms.

3.3.1. Mapping for BPEL The mapping between the BPEL model (more precisely, the relevant parts thereof) and our composition model is depicted in Figure 4.

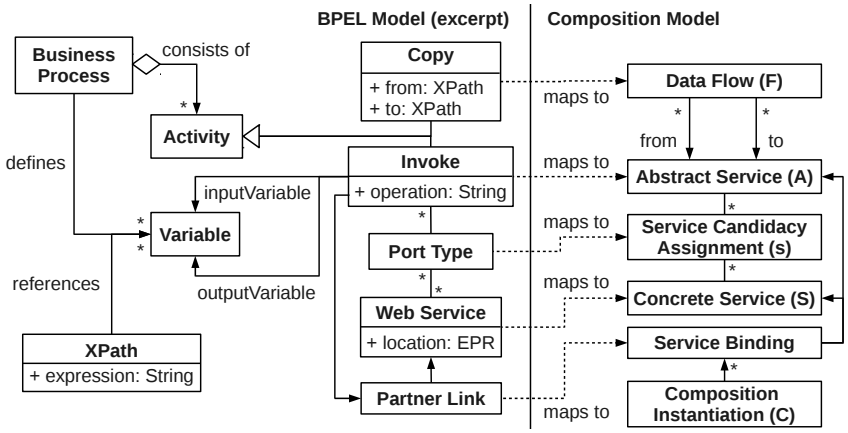

Figure 4. Mapping between BPEL Model and Composition Model

A BPEL *business process* consists of multiple *activities* which define the processing logic. For our purpose, the activities *invoke* and *copy* are most relevant. An *invoke* activity performs a Web service invocation, taking the value of an *input variable* and storing the result to an *output variable*. The *copy* activity uses XPath expressions to process the results received from an invocation and to assign values between output and input variables. In this respect, *invoke* corresponds to abstract service in our composition model, and *copy* represents a data flow. Concrete services are implemented as *Web services* which are addressable using a unique Endpoint Reference [30] (EPR). The *port type* of a Web service defines the set of abstract operations and the messages involved [31]. Hence, the *port type* provides the basis for service candidacy assignment (function $s : A \rightarrow S$) in the composition model. Finally, the *partner link* concept is used in BPEL to model the required relationships between

DOI: 10.1002/stvr

services and partner processes. At runtime, a *partner link* is bound to the location (EPR) of a Web service, which corresponds to the service binding in the composition model.

3.3.2. Mapping for WS-Aggregation The relationship between the WS-Aggregation query model [17, 18] and the composition model is illustrated in Figure 5.

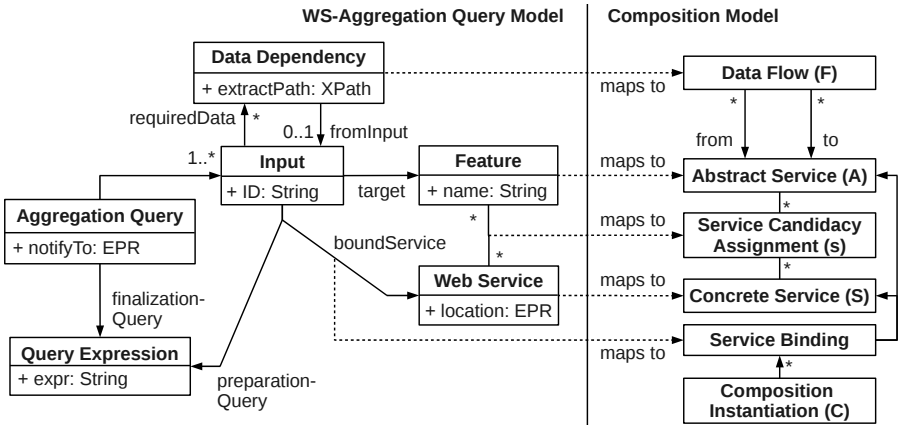

Figure 5. Mapping between WS-Aggregation Query Model [17, 18] and Composition Model

The core artifact in the query model is the *aggregation query*, which consists of multiple *input* elements that represent the data received from event streams of external Web services. The attribute *notifyTo* contains the Endpoint Reference (EPR) of the client that receives the results from the platform in push mode. For each input, a *preparation query* filters, preprocesses and aggregates the received events. A *finalization query* is used to combine the data into a single result document. The distinction of abstract and concrete services is achieved by an association between *features* and *Web services*: each input targets a certain feature (which maps to an abstract service), and each feature is provided by several Web services (which map to concrete services). This corresponds to the service candidacy assignment (function $s : A \rightarrow S$) in the composition model. Moreover, inputs are associated with one or more instances of *data dependency*. A dependency between two inputs i_1 and i_2 indicates that certain data (specified via *extractPath*) need to be extracted from the results of i_2 and inserted into i_1. The internal query processor of WS-Aggregation automatically distributes the execution of inputs among the deployed aggregator nodes, resolves dependencies and takes care of proper correlation of the involved event streams. Clearly, input data dependencies can be mapped directly to data flows in the composition model. Finally, a service binding in our model translates to the runtime binding of a Web service for a specific input in WS-Aggregation.

4. COMBINATORIAL TEST DESIGN

Finding the minimal set of composition test cases, taking into account k-node data flow coverage and all of the model's constraints, is a computational problem in the area of combinatorial test design [32, 33], which is known to be NP-hard [34]. The number of possible compositions is exponential in the number of services. It is simple to construct any valid solution, as well as to determine the validity of existing solutions, however, no polynomial-time algorithm exists that can guarantee to deliver the optimal solution. The size difference between 1) a (near-)optimal solution and 2) a test set obtained by full enumeration (cf. Table I) or a simple construction heuristic can be significant. The implication is that in case of 1) the test generation takes longer and the test set executes faster, whereas for 2) the test set can be generated fast and executing the test takes longer due to the increased size of the solution. Note that we speak of *near*-optimality, because in any case we seek for a practicable approach that executes in acceptable time. The optimization details for finding minimal sets of test cases for service compositions are discussed in the following.

DOI: 10.1002/stvr

4.1. Optimization Target

Our goal is to keep the required testing effort minimal. Hence, the universal objective function attempts to minimize the number of composition test cases to execute, which is expressed in Equation 3.

$$|T| \rightarrow min \tag{3}$$

In addition to this universally valid criterion, it is advantageous to specify preferences concerning service reuse. A composition tester who has knowledge about the quality of the individual services may give precedence to certain concrete services. Applied to the scenario, consider that s_1 is known to be well-tested, whereas s_2 is published by a less reliable provider. We introduce the additional symbol $p(s_x, a) \in (0, 1)$ to specify the desired probability that service s_x should be bound to abstract service a in the test cases (e.g., $p(s_1, a_1) = 0.2, p(s_2, a_1) = 0.8$). Based on that, our alternative objective function minimizes the total deviation between the actual and the expected occurrences of all concrete services in the generated test cases (Equation 4). Note that the special case in which all probabilities are equal, $\forall a \in A : \forall s_x, s_y \in s(a) : p(s_x, a) = p(s_y, a)$, expresses that all services should be tested with the same intensity. Further note that the extreme values 0.0 and 1.0 are not in the value domain of $p(s_x, a)$; the effect of 0% or 100% probability can be achieved by completely removing the service s_x from the candidate list $s(a)$, or by removing all other services from the candidate list, respectively.

$$|T| + \sum_{a \in A} \sum_{s_y \in s(a)} \left| \frac{o(s_y,a)}{|s(a)|} - p(s_y, a) \right| \rightarrow min \tag{4}$$

Equation 3 is the default optimization target, if there is no known reason for preferring the testing of some concrete services over others. If there is a reason for a preference, it can be easily expressed by using probabilities with Equation 4. Both of the alternative objective functions defined above are subject to the following hard constraints:

$$c(a) \neq \emptyset, \forall c \in T, a \in A \tag{5}$$

$$\bigcup_{c \in C, a \in A} c(a) = \bigcup_{a \in A} s(a) \tag{6}$$

$$\begin{aligned} &\forall \quad j \in \{1, .., k\} : \\ &\forall \quad d_j \in F_j : \\ &\forall \quad s^1_{dj} \in s(a^1_{dj}), ..., s^j_{dj} \in s(a^j_{dj}) : \\ &\exists \quad c \in T : c(a^1_{dj}) = s^1_{dj} \wedge ... \wedge c(a^j_{dj}) = s^j_{dj} \end{aligned} \tag{7}$$

Equation 5 signifies that a composition must bind one concrete service to each abstract service. Equation 6 expresses that each concrete service that implements one of the abstract services needs to be used at least once in the final set of compositions. Finally, Equation 7 ensures that the service composition is k-coverage tested, i.e., that all services with direct and indirect data dependencies (up to flows of length k), are tested with all combinations of concrete services. Note that there is a logical connection between the criteria in Equations 6 and 7, as follows. According to Equation 7, the composition is k-coverage tested (for $k \geq 1$). This trivially implies that the composition is 1-coverage tested. Now, 1-coverage tested means that every single abstract service is tested (at least once) with each of its concrete candidates, which corresponds to the semantics of Equation 6. So, in fact Equation 6 is an indirect implication of Equation 7; however, both equations have been included here for clarity.

DOI: 10.1002/stvr

4.2. Determining Faulty Services and Incompatible Configurations

So far, we have discussed that the composition model is used to compute a (minimal) set T of composition test cases. The detailed procedure for generating and executing the tests will be addressed later in Section 5. For now, we assume that the tests have been executed and for each instantiation $c \in T$, a test outcome has been recorded. The test results contain data for non-functional properties, such as the processing time, as well as the functional test data, which informs about whether the test case has succeeded or failed. Recall from Section 3.1 that the functional result of a single test case is expressed as a boolean function $r : C \to \{success, failed\}$.

4.2.1. The Fault Participation and Fault Contribution Metrics Having determined the functional test result of each composition instance, we are able to analyze which service or which combination of services have caused tests to fail. In fact, we are interested not only in detecting the existence of a fault, but in determining faulty concrete services or incompatible combinations. Let $x : A \to S$ denote an assignment of concrete services to abstract services, which represents any subset of the service bindings applied in a composition $c \in C$, that is, $x \subseteq c$. For instance, take the function values of the scenario composition c_4 in set-of-pairs notation, $c_4 = \{(a_1, s_2),(a_2, s_3),(a_3, s_4),(a_4, s_4),(a_5, s_2),(a_6, s_7)\}$. One possible subset of this composition would be a binding $x = \{(a_4, s_4),(a_5, s_2)\}$. Note that x is also a subset of c_3, c_{15} and c_{16}. The function $succ : \mathcal{P}(C) \to \mathcal{P}(C)$ returns for a set of service compositions $Y \in \mathcal{P}(C)$ those for which a successful test result has been recorded: $succ(Y) := \{c \in Y \mid r(c) = success\}$. Analogously, the set of failed test compositions in Y is $fail(Y) := \{c \in Y \mid r(c) = failed\}$. Finally, we use the function $match : ((A \to S) \times \mathcal{P}(C)) \to \mathcal{P}(C)$, $(x, Y) \mapsto \{c \in Y \mid \forall (a, s) \in x : c(a) = s\}$ to determine all compositions in a set Y, which contain all assignments defined by a binding x.

We then utilize the following two indicators:

- The fault *participation* rate $part(x, T)$ of a binding x denotes the percentage of failed compositions in T that contain all bindings defined by x (in relation to all failed compositions in T):
  $part(x, T) := \frac{|fail(match(x,T))|}{|fail(T)|}$
- The fault *contribution* rate $cont(x, T)$ of a binding x denotes the percentage of compositions in T matching x that failed in the test (in relation to all compositions in T matching x):
  $cont(x, T) := \frac{|fail(match(x,T))|}{|match(x,T)|}$

Information Retrieval		Test Result Analysis	
Precision =	$\frac{\#(\text{Relevant Docs in Answer Set})}{\#(\text{Answer Set})}$	**Contribution =**	$\frac{\#(\text{Failed Tests with Assignment } x)}{\#(\text{Tests with Assignment } x)}$
Recall =	$\frac{\#(\text{Relevant Docs in Answer Set})}{\#(\text{Relevant Docs})}$	**Participation =**	$\frac{\#(\text{Failed Tests with Assignment } x)}{\#(\text{Failed Tests})}$

Figure 6. Analogy Between Fault Contribution/Participation and Precision/Recall

The fault participation expresses the degree to which binding x has been "involved" in the faulty compositions. A higher value of $part(x, T)$ indicates that x is likely (partly) responsible for the fault. On the other hand, fault contribution expresses the degree to which the test cases containing x have led to a faulty result. That is, $cont(x, T)$ indicates whether there have been any successful results for tests including x. Note that these indicators are related to *precision* and *recall* [35], two measures often used in machine learning and in information retrieval (IR). We can draw an analogy between document search in IR and finding incompatible bindings in services testing, which is illustrated in Figure 6. The precision of a performed search in IR expresses the fraction of retrieved documents that are relevant with respect to the search criteria. Similarly, fault contribution considers the fraction of test cases that are "relevant", i.e., failed. On the other hand, recall answers the question to which

DOI: 10.1002/stvr

	#	$\mathbf{a_1}$	$\mathbf{a_2}$	$\mathbf{a_3}$	$\mathbf{a_4}$	$\mathbf{a_5}$	$\mathbf{a_6}$	**Result**
Minimal Test Set	$\mathbf{c_4}$	s_1	s_3	s_1	s_1	s_2	s_8	Failed
	$\mathbf{c_{13}}$	s_2	s_3	s_4	s_4	s_2	s_7	Succeeded
	$\mathbf{c_{17}}$	s_1	s_3	s_5	s_5	s_2	s_8	Failed
	$\mathbf{c_{20}}$	s_2	s_3	s_1	s_1	s_6	s_8	Succeeded
	$\mathbf{c_{21}}$	s_1	s_3	s_4	s_4	s_6	s_8	Succeeded
	$\mathbf{c_{12}}$	s_2	s_3	s_5	s_5	s_6	s_7	Succeeded
Additional Tests	$\mathbf{c_2}$	s_2	s_3	s_1	s_1	s_2	s_7	Succeeded
	$\mathbf{c_{10}}$	s_2	s_3	s_4	s_4	s_6	s_7	Succeeded
	$\mathbf{c_{16}}$	s_2	s_3	s_4	s_4	s_2	s_8	Failed

Table IV. Illustrative Test Results of Scenario Composition Test Cases T (cf. Table I)

degree the relevant documents have been found by a search. Likewise, fault participation relates the failed test results of a certain service assignment to the set of all relevant (failed) test cases. To obtain a one-dimensional measure, precision and recall are often combined into a harmonic mean, denoted *F measure* [35] (or *F score*), which is also conceivable as an alternative for $part$ and $cont$.

Table IV lists example test results for different test cases of the scenario composition, based on which we discuss the relevance of the indicators $part$ and $cont$ in more detail. The first six entries in the table represent the minimal set of test cases that is required to satisfy the k-node data flow coverage criterion (k=3). For illustration purposes, we have injected an incompatibility between the concrete services s_2 and s_8 with respect to the data flow between the abstract services a_5 and a_6. The goal is to detect this incompatibility based on the test results (*Failed* or *Succeeded*) of the individual test cases, which is illustrated in Table V. This table contains the values for $part$ and $cont$ for six randomly chosen sample bindings. The number of bindings for which we have to compute the indicators depends on the number of abstract and concrete services, as well as the number and length of the composition's data flow paths. In the case of our example, 34 distinct combinations of bindings exist (13 bindings for single services, 15 bindings along the four different 2-node data flows, and 6 bindings along the 3-node data flow). Six out of the 34 binding combinations are printed in Table V. The most interesting cases are those for which $cont$ equals 1, which in this example occurs, as expected, only for the faulty service combination $x : a_5 \mapsto s_2, a_6 \mapsto s_8$. Additionally, for this combination also the value of $part$ is 1, because the example contains only a single faulty service combination, and hence no other combination participates in all faulty test cases. Technically, the binding $x : a_2 \mapsto s_3, a_5 \mapsto s_2, a_6 \mapsto s_8$ also has a value of 1 for both indicators, which results from the fact that s_3 is the only concrete service available for a_2. Our evaluation in Section 6.4 discusses how the test indicators evolve if multiple service combinations are causing faults in a composition.

Combination x :	$a_1 \mapsto s_1$	$a_2 \mapsto s_3$	$a_5 \mapsto s_2$	$a_1 \mapsto s_2$, $a_2 \mapsto s_3$	$a_5 \mapsto s_2$, $a_6 \mapsto s_8$	$a_1 \mapsto s_2$, $a_3 \mapsto s_1$, $a_4 \mapsto s_1$	...
Values for Minimal Test Set:							
$\mathbf{part(x, T)}$	1.0	1.0	1.0	0.0	1.0	0.0	...
$\mathbf{cont(x, T)}$	0.6667	0.3333	0.6667	0.0	1.0	0.0	...
Values for Minimal Test Set and Additional Tests:							
$\mathbf{part(x, T)}$	0.6667	1.0	1.0	0.3333	1.0	0.0	...
$\mathbf{cont(x, T)}$	0.6667	0.3333	0.6	0.1667	1.0	0.0	...

Table V. Exemplary Service Combinations for the Scenarios (6 out of 34 Total Combinations) with Fault Participation and Fault Contribution

4.2.2. Additional Tests Beyond the k-Coverage Minimal Test Set The described technique is suitable to automatically analyze the test results and point the tester towards faults in the service composition. Note that the discussed indicators depend on the number of test results at hand. In general, a higher number of test cases provides stronger evidence. For instance, after executing the first test composition (c_4), which failed, trivially all service assignments of this composition have

Prepared using stvrauth.cls
DOI: 10.1002/stvr

participated in and contributed to 100% of the faults. Table V also illustrates that the values change when additional tests (c_2,c_{10},c_{16}) are executed on top of the minimal test set with respect to k-node test coverage. For instance, $part(x, T)$ is 1.0 for $x : a_1 \mapsto s_1$, and the minimal test set of six compositions, but we can exclude x from the potentially problematic assignments as the value drops to 0.6667 after executing the additional tests. Conversely, the assignment $x : a_1 \mapsto s_2, x : a_2 \mapsto s_3$ is not associated with any failed results in the minimal test set, but shows a failed result in the additional tests. This example shows that the minimal test set is generally sufficient for a meaningful test result analysis, but evidently additional test data leads to more precise localization possibilities. In any case, the coverage criterion ensures that all crucial service combinations are tested with respect to direct and transitive data dependencies.

4.2.3. Alternative and Advanced Fault Localization Techniques It should be noted that by means of the presented contribution/participation method, it is easier to detect a single data flow incompatibility than to analyze faults in compositions that contain several combinations of incompatible services. The reason is that if multiple incompatibilities exist, the $part$ or $cont$ values will likely never reach a value of 1, and these cases require the tester to define a threshold value or to investigate those service combinations with high $part$ or $cont$ values manually. This aspect is evaluated in more detail in Section 6. Moreover, this method of test result analysis is reliable only if we can assume that faults happen deterministically, in the sense that any faulty service assignment x always leads to a failed result in all test cases it participates in: $|fail(match(x, T))| > 0 \Rightarrow |match(x, T)| = |fail(match(x, T))|$. Formulated differently, this means that all tests have to be repeatable. We believe that this is a valid working assumption for this paper, as the main contribution here is not to discover failures in services themselves, but to find problems originating in the data flow between services.

Fault detection and localization is an active research field that continues to produce sophisticated ongoing results. For instance, approaches in the area of software fault localization (e.g., [36, 37, 38, 39]), whose primary target are faults/bugs on the software source code level, are partly applicable to service compositions as well. In particular, the metrics termed *hue* and *suspiciousness* in [37] are related to the fault participation and fault contribution metrics defined here. Another related fault localization method has recently been applied in [40], which, similar to our approach, uses coverage metrics based on information flow. The approach ranks source code statements regarding their likelihood of being faulty, and this ranking is *"primarily determined by contrasting the percentage of failing runs to the percentage of passing runs that induced it"* [40]. We would also like to point out recent work in [41, 42], which studies machine learning techniques for advanced localization of incompatible service implementations in service compositions. For this purpose, a generic service composition model is established which also considers the values of the data that flow between services. In order to limit the search space, which grows even larger when considering the data flow values, custom partitioning is applied to the data value domains [42]. For this paper, however, we focus on the service composition level and employ an input data-agnostic approach to fault localization. Moreover, we point out that the fault localization employed here has the main focus of evaluating the coverage achieved by our approach for testing dynamic service compositions.

5. IMPLEMENTATION: THE TECOS FRAMEWORK

The work of this paper is integrated into the *TeCoS* (*Te*st *Co*verage for *S*ervice-based systems) framework, which we briefly present in the following. TeCoS aims at providing a holistic view of SBAs by providing various coverage metrics, thereby operating both on the level of the service API (Application Programming Interface) and on the service composition level. In previous work, we have presented how TeCoS collects and stores invocation messages at runtime in order to compute API coverage metrics of single services [43], whereas in this paper the focus is on achieving test coverage for composed services. The task of TeCoS is to provide the data necessary to construct the composition model, as well as to generate and execute the composition test cases. Figure 7 sketches the TeCoS architecture in the context of the service composition implementing our scenario. The

DOI: 10.1002/stvr

Web services are provided and hosted by three service providers. A service integrator defines and publishes the composition on top of the services, e.g., as a BPEL process or a WS-Aggregation query. End users invoke the composition, and may also make use of the atomic services $(s_1, ..., s_8)$.

The TeCoS Service Broker is a centralized entity that offers a Service Registry to store service metadata, and a Tracing Service which is responsible for logging invocations that occur in the SBA, both for the atomic services and for the BPEL process. Our implementation provides an invocation interceptor that is easily plugged into the Java Web services framework. The traces and coverage data can be inspected via the Web User Interface. Here, we do not consider data protection and privacy issues in much detail, but service providers and integrators may choose to host their own instances of the Tracing Service and receive events about newly added data from the broker. For privacy and access control related issues in service compositions, we direct the interested reader to our related previous work [26]. The core component, which orchestrates the testing process is the Test Manager (TM). The TM repetitively invokes the Composite Web Service (WS), and each repetition covers one test case. For instance, if the Composite WS is implemented in BPEL, the process definition is instrumented in such a way that it dynamically retrieves from the TM the Endpoint Reference (EPR) of the services to invoke (see Section 5.4). A service EPR uniquely identifies a service instance and contains the technical information required to invoke the service, such as the service name and its location URL (Uniform Resource Locator).

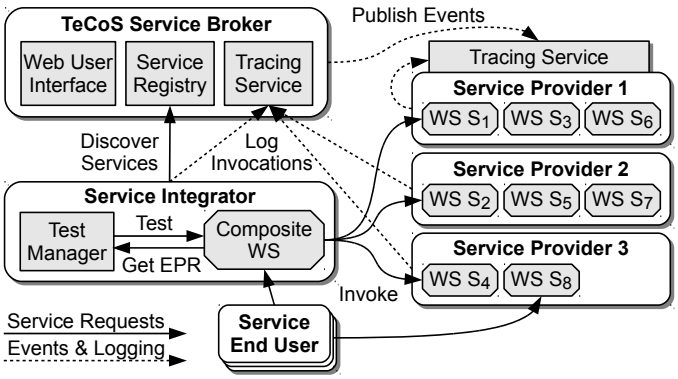

Figure 7. TeCoS Architecture

TeCoS logs every execution of the service composition in order to collect traces that can be used in the FoCuS solver as described in Section 5.3. A crucial point about logging a composite service is to correlate the single invocations performed by one of its runtime instances. Consider two users invoking the BPEL process simultaneously. The order of the subsequent service invocations performed by the two process instances is indeterministic, and hence the exchanged messages need to carry additional information to allow for identification of process instances. Similar to the solution proposed in [44], we use BPEL code instrumentation to inject unique identifiers in the exchanged messages. SOAP (Simple Object Access Protocol) is the messaging protocol used by Web services. Whereas the SOAP *body* carries the payload of the message, the SOAP *header* is used to transmit additional information. We use the SOAP header to include the identifier of the BPEL process instance in each invocation it carries out.

In the following we describe the implemented end-to-end solution in more detail, particularly focusing on how test cases are generated and executed in BPEL and WS-Aggregation. Section 5.1 shows how target composition platforms can be plugged into TeCoS by means of an adapter mechanism. To further discuss the details of the single steps that have been outlined earlier in Figure 1, we divide the procedure of the end-to-end approach into three parts: the test preparation activities before the optimization takes place (Section 5.2), the activity where FoCuS obtains the optimized solution (Section 5.3), and the part after optimization where the tests are generated, executed and analyzed (Section 5.4).

DOI: 10.1002/stvr

5.1. Integration of Target Composition Platforms via Extensible Adapter Mechanism

The testing approach and coverage criterion proposed in this paper are applicable to various service composition techniques. The basic requirement is that the composition model needs to provide a mapping between abstract and concrete services and that compositions can be instantiated with a particular service assignment. Two sample techniques, BPEL and WS-Aggregation, have been exemplified, and as part of our ongoing work we are integrating additional platforms. For instance, in the emerging field of service mashups [45] the Enterprise Mashup Markup Language [46] (EMML) is one recommended way of building light-weight service compositions. Another example is Windows Workflow Foundation[¶], a programming model for service-based business processes.

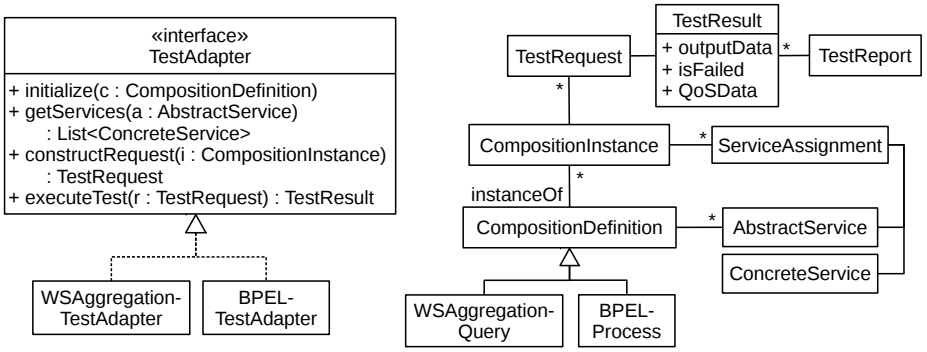

Figure 8. Model for Platform-Specific Composition Test Adapters

TeCoS provides a flexible adapter mechanism to map the results of combinatorial test design to concrete composition platforms. The model for these platform-specific test adapters is depicted as a UML class diagram in Figure 8. The core interface is `TestAdapter`, which defines methods to initialize an adapter for a certain composition definition (`initialize`), to determine the list of candidate services for an abstract service (`getServices`), to construct a request for a certain composition instance (`constructRequest`), and to execute a single test request (`executeTest`). The final test report is composed of one test result for each composition instance, containing the output data, a boolean flag indicating whether the test has failed, and a collection of QoS metrics that are particularly interesting for testing the quality of event-based compositions. Specialized adapters implement the interface `TestAdapter` and provide an extension of the `CompositionDefinition` class. The remaining classes in Figure 8 are agnostic of the concrete target platform and define the general model of compositions with dynamic service assignment.

5.2. Test Preparation Steps

The first step in the test preparation is to create the data flow view from the composition definition, which in our scenarios are a BPEL document and a WS-Aggregation query, respectively. In WS-Aggregation, the data flow view can be directly derived from the user request. Each composition query is composed of multiple inputs (which correspond to the abstract services) that receive data from concrete services [17]. The query model allows to define data dependencies between inputs which directly map to the data flows between services as considered in this paper.

```
1 <bpel:assign xmlns:bpel="...">
2  <bpel:copy>
3   <bpel:from>$a3_out.part1//hotel[$i]</bpel:from>
4   <bpel:to>$a4_in.part1//hotel</bpel:to>
5  </bpel:copy>
6 </bpel:assign>
```

Listing 1: Data Dependency in BPEL Variable Assignment

[¶]http://msdn.microsoft.com/en-us/netframework/aa663328.aspx

DOI: 10.1002/stvr

For BPEL, all `assign` activities are analyzed to filter those assignments that copy from an invocation input to an invocation output variable. To detect indirect assignments via auxiliary variables, the `assign` activities are analyzed recursively. A sample `assign` activity is printed in Listing 1, `$a3_out` and `$a4_in` denote the variable names pointing to the input and output messages, followed by a dot (`"."`) and the target WSDL *message part*. We can ignore all assignments which do not create an inter-invocation data dependency (e.g., those that assign constant values, increase counter variables etc). As we know the pattern for variable names, `$a3_out` and `$a4_in` can be extracted using regular expressions and we have determined a data dependency. Note that this method only works reliably if every invocation in BPEL has its own pair of input/output variables. This is usually the case and otherwise a warning message is issued.

The second test preparation step is the determination of the concrete candidate services. The service registry contains references to available Web services and their WSDL interface descriptions. The WSDLs are retrieved and parsed and we loop over all `invoke` activities in the BPEL process: a service becomes a candidate if it implements the `portType` required by an invocation. In WS-Aggregation, the definition of abstract and concrete services is directly incorporated into the query model. Each query input (i.e., abstract service) targets a certain *feature* which is basically a string describing the capability of the service that has to provide this input. The service registry contains the mapping between features and concrete candidate services.

The third and last preparatory step is to combine all the gathered information and transform it into the FoCuS data model. As mentioned earlier in this section, TeCoS provides the logging data of previous executions of the composition under test. From the pool of logged invocations we filter those that carry the same process instance identifier (ID) in the SOAP header, and provide this data as FoCuS traces.

5.3. Transformation to FoCuS Data Model

FoCuS [15] implements algorithms for combinatorial test design and is used for optimization of our target function in Section 4.1. Note that FoCuS is only one possibility to achieve the optimization step in our end-to-end approach. We have also experimented with other techniques such as Constraint Programming [47] using the Choco[‖] solver, but FoCuS has turned out to perform best, even for larger scenarios. In the following, we describe the mapping from the service composition model to the data model used by FoCuS (see Table VI).

Service Composition Model	**FoCuS Model**
abstract service	attribute
concrete service	value
interrelated services	restriction
incompatible services	restriction
occurrence precedence	attribute weights
uniform service reuse	attribute weights (=1)
existing execution	trace

Table VI. Mapping from Service Composition Model to FoCuS Model

FoCuS uses the notion of *attributes* and *values*. Each abstract service in our model maps to an attribute in FoCuS. The values of the attributes are the concrete services that implement the required interface. To that end, each service is identified by a unique integer number.

FoCuS allows to impose custom *restrictions* on the attributes values. We use restrictions to express that two or more services should 1) never or 2) always be used together. For instance, in the presented scenario, the services a_3 and a_4 are interrelated in the sense that they should be executed by the same concrete service instance, so we add a FoCuS restriction for the attributes' equality: `a3 == a4`. For service combinations that are known to be incompatible (and should hence not occur together in a test case), we add inequality restrictions.

[‖] http://www.emn.fr/z-info/choco-solver/

DOI: 10.1002/stvr

Constraints concerning the uniform reuse of services or precedence of certain service instances (Equation 4) can be expressed in FoCuS using *attribute weights*. Weights express the ideal distribution of attribute values, and are a possibility to influence the value computation. Expressed in terms of the service composition model, if the candidate services $s(a) = \{s_a^1, s_a^2, .., s_a^j\}$ of an abstract service a have weights $w(s_a^1), w(s_a^2), .., w(s_a^j) \in \mathbb{R}$, then the (ideal) probability that some service s_a^x is chosen for testing the abstract service a is $\frac{w(s_a^x)}{\sum_{s_a^y \in s(a)} w(s_a^y)}$. Note that FoCuS gives no guarantee about value distribution, but weights are taken into consideration during optimization.

Finally, FoCuS offers the possibility to provide data about existing previous *traces*. Traces constitute a subset of solutions that should not (or need not) be considered. The algorithm then attempts to cover all remaining value combinations. Adding traces can help reduce the complexity of the problem and the runtime of the algorithm. Section 5 gives more information about how the traces are collected.

5.4. Generating & Executing Tests

After FoCuS has finished generating a feasible and near-optimal solution for the given model, the automated test generation starts. The goal is to prepare the composition under test to bind to the services determined in the test design. The solution in WS-Aggregation is straight-forward, as the query model allows to specify the concrete services to be used along with the request. That is, for each of the generated test cases a request reflecting the respective service assignment is sent to the platform. In BPEL, hardcoding the concrete services in the process definition itself would require the deployment of a large number of process instances (as many instances as test cases to be executed), which is time-consuming and often infeasible for large test sets. Therefore, we aim at deploying only a single process instance that is able to dynamically look up and bind to the correct EPRs at runtime. This is achieved by instrumenting additional commands into the process definition. The corresponding algorithm is sketched in Algorithm 14. To illustrate the structure of an instrumented BPEL process, the rough skeleton of an example process is printed in Listing 2 (dots ".." in the listing indicate that parts have been left out for clarity).

```
<process>
 <import location=".." ../>
 ..
 <partnerLinks>
  <partnerLink name="TMS" ../>
  ..
 </partnerLinks>
 ..
 <variables>
  <variable
     name="instanceID" ../>
  <variable name=".." ../>
  ..
 </variables>
 ..
 <sequence>
  <assign name="a1">..</assign>
  <invoke operation="getEPR"
     partnerLink="TMS" ../>
  <assign name="a2">..</assign>
  <assign name="a3">..</assign>
  <invoke ../>
 </sequence>
 ..
</process>
```

Listing 2: Instrumented BPEL Process

1: add `import` elements for WSDL and XSD imports
2: `tms` ← new `partnerLink` for Test Manager Service
3: `instanceID` ← new `variable`, initialized as GUID
4: **for all** `invoke` activities `i` **do**
5: `pl` ← `partnerLink` of invocation `i`
/ First, add statements to request EPR from Test Manager Service and to set dynamic partner link. */*
6: define `variables` $\texttt{eprIN}_\texttt{i}$ and $\texttt{eprOUT}_\texttt{i}$
7: add `assign` $\texttt{a}_1$: $\texttt{eprIN}_\texttt{i}$ ← name of `pl`
8: add `invoke` $\texttt{i}_1$: $\texttt{eprOUT}_\texttt{i}$ ← `tms.getEPR`($\texttt{eprIN}_\texttt{i}$)
9: add `assign` $\texttt{a}_2$: `pl` ← $\texttt{eprOUT}_\texttt{i}$
/ Second, add statements to set BPEL instance ID in SOAP headers for invocation `i`. */*
10: `hdr` ← new header element for invocation `i`
11: add `assign` $\texttt{a}_3$: `hdr` ← `instanceID`
12: `s` ← new `sequence`: $\texttt{a}_1 \,||\, \texttt{i}_1 \,||\, \texttt{a}_2 \,||\, \texttt{a}_3 \,||\, \texttt{i}$
13: replace invocation `i` with sequence `s`
14: **end for**

Algorithm 1: BPEL Instrumentation Algorithm

DOI: 10.1002/stvr

After adding the required global definitions and generating a Globally Unique Identifier (GUID) in lines 1-3, the algorithm loops over all `invoke` activities and ensures 1) that the correct EPR for this invocation is retrieved from the Test Manager Service (lines 6-9), and 2) that the process ID is sent along with the invocation (lines 10-11).

The instrumented BPEL process is then deployed to the target BPEL engine. Via the `getEPR(String partnerLinkName)` service method the Test Manager (TM) provides the EPR information according to the current test case. The mapping between abstract and concrete services is stored in a service registry, implemented using the VRESCo SOA runtime environment [48]. The TM uses the specified test inputs and repetitively executes the BPEL process. The result of each composition execution is matched against the expected output, which is specified by the composition developer (service integrator) in the form of XPath expressions.

5.5. *Test Oracle*

Test oracle [27] denotes the mechanism which determines whether a result is acceptable and whether a test case has failed or passed. In the case of synchronous (request-response) compositions, the test oracle is defined as a set of assertions over the result data returned by the composition. In the current implementation of TeCoS these assertions are defined as XPath expressions that are required to evaluate to a positive Boolean result (*true*). The oracle evaluates the XPath expressions and simply compares the result to the expected output defined by the tester.

The test oracle becomes more sophisticated for event-based continuous queries. The correct functioning of compositions in WS-Aggregation depends not only on a single result document, but on the timing and sequence of multiple arriving results. Hence, the assertions are defined and evaluated over a multitude of data records. In fact, the oracle definition can itself be seen as a query over the sequence of received result documents, and this sequence has to fulfill certain criteria. For instance, in our scenario we expect that, over time, hotel room availability and flight data will be available for each hotel and location provided to WS-Aggregation. Moreover, we can assume that the final results arrive with roughly the same frequency as the highest frequency of any of the event inputs. If no result is received for a long time period, the tester would define this as an indicator that the composition test case is failed.

6. EVALUATION

To evaluate different performance aspects of the proposed approach, we have set up an experimental evaluation in a private Cloud Computing environment with multiple virtual machines (VM), managed by an installation of *Eucalyptus*[**].

Our experiments focus on four aspects: firstly, the effect of the k-node coverage criterion on the size of the test case set (Section 6.1); secondly, the performance of testing WS-BPEL instances of different sizes (Section 6.2); thirdly, the performance of testing long-running and event-based compositions in WS-Aggregation (Section 6.3); fourthly, the effectiveness of identifying incompatible service assignments and the evolution of the respective indicators over time (Section 6.4).

The experiments in Sections 6.1 and 6.2 have been run on a single machine with Quad Core 2.8GHz CPU, 3GB memory, running Ubuntu Linux 9.10 (kernel version 2.6.31-22). For the distributed performance tests in Sections 6.2.1 and 6.3, multiple cloud VM instances with slightly weaker performance characteristics have been used. Each VM is equipped with 2GB of main memory and one virtual CPU core with 2.33 GHz (comparable to the *small* instance type in Amazon EC2[††]).

[**] http://www.eucalyptus.com/
[††] http://aws.amazon.com/ec2

DOI: 10.1002/stvr

6.1. *Effect of k-Node Coverage Criterion on the Number of Test Cases*

First, we investigate the effect of the k-node data flow coverage criterion on the number of generated test cases. Consider three imaginative composition scenarios (S=small/M=medium/L=large), see Table VII. S/M/L have, respectively, 6/10/20 abstract services, with 5/10/20 concrete services per abstract service, and different data flows of length 2/3/4.

	Small		**Medium**		**Large**	
Abstract Services	6		10		20	
Concr. S./Abstr. S.	5		10		20	
2-Node Data Flows	2		5		10	
3-Node Data Flows	1		2		5	
4-Node Data Flows	-		1		2	
Min. $\|T\|$ for $k=4$	125		10000		160000	
Min. $\|T\|$ for $k=3$	125		1000		8000	
Min. $\|T\|$ for $k=2$	25		100		400	
Min. $\|T\|$ for $k=1$	5		10		20	
FoCuS Results	min	max	min	max	min	max
Execution Time (seconds)	0.15	0.48	2.91	3.84	742.9	939.3
Test Cases ($\|T\|$)	125	125	10K	10K	160003	160008
Occurrence Differences	35	48	557	623	5148	5935

Table VII. Optimization of Different Model Sizes

The lower bound of generated test cases, $min(|T|)$, varies with the value of k. For the special case of $k=1$, $min(|T|)$ equals the maximum number of concrete services per one abstract service, $max(|s(a_x)|), a_x \in A$. In general, $min(|T|) = max(|s(a_x)|)^k$, provided that the composition contains any data flow of length k. Note that these bounds are only valid if 1) the composition is free of constraints (e.g., that some abstract services must or must not bind to a specific concrete service), and 2) no previous execution traces exist. We applied the FoCuS optimization to the three scenario sizes (see Table VII). The FoCuS optimization was repeated 20 times and the table lists the minimum and maximum execution times (seconds), the number of generated test cases and the total service occurrence differences ($\sum_{a \in A}(max(a) - min(a))$). $|T|$ of the FoCuS result is optimal for S and M, and close to $min(|T|)$ for L.

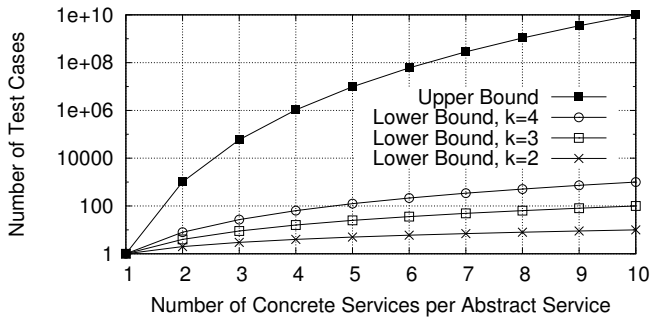

Figure 9. Combinations in Medium Scenario (cf. Table VII)

Figure 9 illustrates the upper and lower bound of $|T|$ in the Medium scenario with increasing number of concrete services. The upper bound represents the number of all possible combinations, which reaches the (infeasible) value of 10^{10} (note the logarithmic scale of the y-axis). Applying the k-node data flow coverage goal ($k \in \{2, 3, 4\}$) drastically decreases the lower bound of the number of test cases. For instance, 10 individual tests need to be executed for a coverage of k=1. Covering k=2 requires 100 test cases, and for k=3 we need data from at least 1000 tests. Arguably, the reduction in the lower bound on the number of test cases delivered by the k-node approach is not surprising, because that approach limits data flow path length. Although this argument is valid, it does not invalidate our approach and its importance, as the k-node approach is very effective in practice, as we demonstrate across this section.

DOI: 10.1002/stvr

6.2. *Performance of Testing BPEL Service Compositions*

To evaluate the end-to-end performance of our testing approach, we have implemented and tested the scenario presented in Section 2. The scenario BPEL process is deployed in a *Glassfish*[‡‡] server. To simulate the services, we make use of *Genesis* [49], a test bed generator for Web services. We generated (pseudo-)random data for the composition input (*date* and *city* parameters in the scenario) as well as for the output of the individual services. For each input and output parameter, values are chosen randomly from a set of predefined possible values. However, in our setup the actual values are only relevant in those cases where data incompatibilities are injected, e.g., different formats for the geographic coordinates, as discussed in Section 2.1.

Table VIII lists the execution time of each step in the end-to-end testing lifecycle (all values are in milliseconds). Firstly, creating the data flow view from the composition definition (BPEL file and WS-Aggregation query) took 151ms. This duration depends mainly on the number of `Assign` tasks contained in the BPEL process. Finding the candidate services took 1205ms. This value depends on the number of services in the registry. We had 30 registered services, and 13 were identified as candidates (see scenario). Converting the composition model to FoCuS took 13ms, and the FoCuS algorithm terminated after 644ms. Initialization of WS-Aggregation finished after 353ms. The largest part of the preparation is BPEL instrumentation and deployment (around 10 seconds).

Test Preparation		**#**	**Dynamic EPR**	**Process Logic**	**Total**
Data Flow View	151	c_4	240.0	408.0	648.0
Find Candidates	1205	c_{12}	280.0	390.0	670.0
Convert Model	13	c_{13}	410.0	331.0	741.0
FoCuS Solver	644	c_{17}	250.0	487.0	737.0
Initialize WS-Aggregation	353	c_{21}	320.0	303.0	623.0
Deploy BPEL	10275	c_{20}	230.0	386.0	616.0
Sum	12641	Avg	288.3	384.2	672.5

Table VIII. Performance of Test Scenario (durations in milliseconds)

The other values on the right side of the table are time measurements of the six test cases generated. To compute the BPEL overhead caused by the instrumented code, we slightly extended the scenario process to have it measure the timestamps before and after the execution of each service invocation. These timestamps are sent to the Test Manager Service at the end of the execution to calculate test case duration end-to-end. The total time is comprised of two elements: setting dynamic EPR according to the test case, which took 288.3ms on average; executing the actual business logic, which depends on the domain, and in our case was on average 384.2ms. The average total time per test case was 672.5ms. Note that the "Process Logic" execution time is scenario-specific, whereas the "Dynamic EPR" time is near-constant and does not depend on the process logic.

6.2.1. Large-Scale Execution of BPEL Tests Whereas the six test cases of our scenario (see $c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}$ in Section 6.2) can easily be executed sequentially, larger test suites require a parallel execution strategy. In the following we evaluate parallel execution of BPEL composition tests in TeCoS. Again, we use our scenario composition definition, but this time we leave out the WS-Aggregation part (a_4 and a_6 are implemented in BPEL and the composition does not wait for 20 seconds), and we provide 10 concrete services per abstract service (to increase the size of the testing problem). Moreover, we drop the requirement that a_3 and a_4 must always bind to the same concrete service. FoCuS generated a minimum of 1000 test instances for this problem (this value corresponds to all combinations of the 3-node data flow in the scenario). We executed these tests in different distributed settings with varying number of w parallel test worker clients ($w \in \{10, 30, 50, 100, 150, 200\}$). A worker is a client program that invokes the service composition under test. The clients are distributed among multiple compute nodes in our Cloud Computing environment; each node hosts 10 clients. Table IX breaks down the test processing times for

[‡‡]https://glassfish.dev.java.net/

DOI: 10.1002/stvr

deployment and actual execution of each test case. The total duration of a test case consists of the actual BPEL process logic plus the overhead for dynamically exchanging EPRs with the TeCoS test manager service. For space reasons, we have only included the values for $w \in \{10, 30, 50, 100\}$, but the remaining results for $w \in \{150, 200\}$ are plotted in Figure 10.

	10 Clients			30 Clients			50 Clients			100 Clients		
	min.	avg.	max.	min.	avg.	max.	min.	avg.	max.	min.	avg.	max.
Deploy BPEL	10103	10103.0	10103	10112	11009.7	12723	9961	10500.0	12313	9362	10196.4	12326
Process Logic	367	795.8	4699	316	1292.0	4120	396	1429.3	2736	390	1830.1	3089
Dynamic EPRs	50	278.4	490	40	447.8	1490	40	469.1	1290	50	580.8	1270
Test Case Total	607	1074.2	4749	356	1739.8	4600	446	1898.4	3246	460	2411.0	3789
Test Suite Total	157057			110760			78334			58375		

Table IX. Performance of Distributed Test Execution (Durations in milliseconds)

As expected, the average deployment time of the process is almost the same regardless of the number of active clients. The fluctuation in the duration of the process logic is application dependent, but in our case the services used by the process are implemented to facilitate a high level of concurrency. That is, a single execution of the process takes roughly the same amount of time in all settings (around 800-1800 milliseconds on average). The slight fluctuations of processing time in the business logic mostly depend on the caching/optimization strategy of the BPEL engine (the first request usually takes longer than the requests to follow). The assignment of dynamic EPRs is fastest in the case of 10 clients, because of the synchronization overhead required when multiple clients access the TeCoS test manager service for EPR information. However, this overhead does not seem to grow strongly (the difference between 30, 50 and 100 clients is negligible).

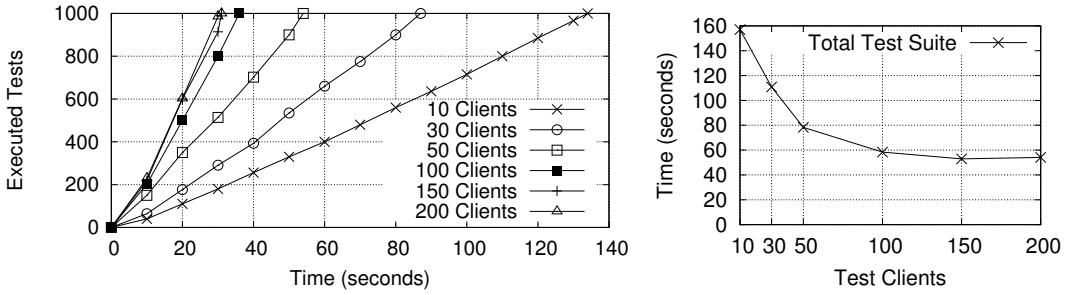

Figure 10. Performance of Distributed Test Execution

A core observation in Table IX and Figure 10 is the degree by which the execution becomes faster as additional test clients are added. The left plot in Figure 10 depicts the number of executed tests against the time required for execution, for different numbers of parallel test clients. We can observe that deploying additional clients is especially beneficial when only few clients are active. For instance, executing the 1000 tests took 133 seconds for 10 clients, but only 87 seconds for 30 clients and 54 seconds for 50 clients. However, the marginal time benefit of adding another client decreases as the total number of clients increases, up to the point where additional clients have no effect at all, which is the case in our experiment when moving from 150 to 200 clients. The reason for this is that the composed services allow only a certain level of parallel requests. Besides faster execution of the total test suite, parallelized test execution has the benefit that the composition is, to a certain degree, stress tested (or load tested), which is a key issue for service compositions and applications that serve multiple concurrent users. However, this is not the core focus of our approach, and systematic approaches for stress testing have been discussed elsewhere (see, e.g., [50]).

6.3. Performance of Testing Event-Based Service Compositions with WS-Aggregation

In the following, we discuss characteristics of testing composite services with continuous and event-based processing using the WS-Aggregation framework. The continuous event-based composition

DOI: 10.1002/stvr

results in WS-Aggregation illustrate nicely the way in which the test indicators evolve over time. To that effect, we implemented the scenario composition discussed in Section 2, but with two minor modifications: firstly, to increase the size of the problem search space (from 24 to 4096 possible composition instantiations), 4 candidate services are provided for each of the 6 abstract services, and the requirement that a_3 and a_4 need to bind to the same concrete service is dropped; secondly, to illustrate more long-running compositions, the WS-Aggregation queries are *not* terminated after 20 seconds, but keep running until the complete test suite is finished. We then deliberately inject faults and incompatible service bindings, execute the test cases, and measure various test indicators, which are discussed in the following. The following results are obtained from a composition with two fault combinations along data flow paths of length 2 ($\{a_5 \mapsto s_2, a_6 \mapsto s_7\}$ and $\{a_1 \mapsto s_1, a_2 \mapsto s_3\}$) and one fault combination along a data flow path of length 3 ($\{a_1 \mapsto s_1, a_3 \mapsto s_1, a_4 \mapsto s_1\}$).

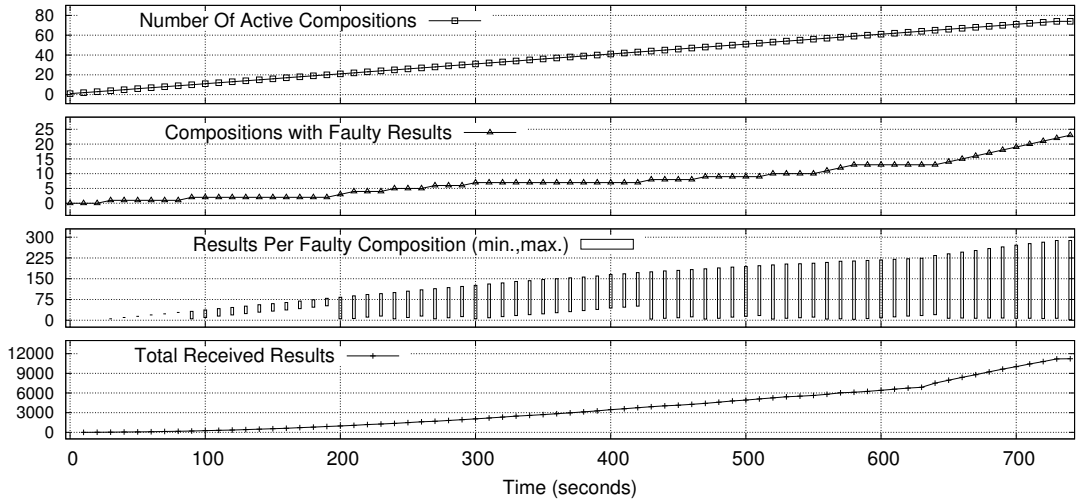

Figure 11. Test Results for the Event-Based Scenario Service Composition

The minimal test case set computed by FoCuS contains 64 combinations, and our experiment successively starts one test case after the other, leaving 10 seconds time between each two test cases. On the one hand, this interval gives WS-Aggregation time to initialize and distribute the execution to the available computing nodes, and on the other hand this repeated procedure allows us to take a snapshot after each added test case and to compare the values. Figure 11 shows how the number of compositions and results evolves over time. The values presented in the figure are taken from one typical and representative end-to-end test run. We do not use averaged values over multiple distinct and isolated test runs because the combinations computed by FoCuS are nondeterministic and the results discussed here depend on the order in which the specific test cases are added to the system.

Figure 11 contains four plots that illustrate the characteristics of a typical test execution of an event-based continuous service composition. The uppermost plot shows how the number of active compositions increases as one instance is added every 10 seconds. Note that the test suite executes more than the minimal set of 64 compositions, because we have added 10 additional instances (i.e., in total 74 test cases are executed) to evaluate the effectiveness of test result analysis later in Section 6.4. The lowermost plot in Figure 11 draws a trendline of the total number of results that have been received as event notification messages from the composition. This curve climbs slowly in the beginning, but the slope gets steeper as more compositions are active. At time point 740 roughly 11000 result events have been received in total. The second plot from the top of the figure contains the number of compositions from which the TeCoS framework has received faulty results, i.e., results with unexpected output data. The third plot from the top depicts the range of results per faulty composition. This value is zero at the beginning (until second 20) and starts to rise as the first faulty composition is activated at time point 30. Up to time point 130, the minimum and maximum are equal (because there is only one faulty composition), and from second 90 onwards the minimum and maximum span up an actual interval range (printed as a box in the figure).

DOI: 10.1002/stvr

Computing and analyzing the number of received composition results (both correct and faulty) is important to obtain a measure for performance-related QoS data. Not shown in the figure are the QoS metrics that are contained in the TeCoS test report, some of which are: *event throughput* (average number of received events per time unit), *regularity* (fluctuations in the interval of received events) or *duplication* (whether or not duplicate events haven been received). These metrics allow a composition tester not only to identify incompatible service combinations, but to favor one composition configuration over another for QoS reasons. A detailed discussion of QoS aspects is out of scope of this paper, and more details can be found, e.g., in [51, 52]. For further details concerning the performance and scalability of WS-Aggregation we refer to [17, 19, 18].

6.4. *Effect of the Measures used to Determine Incompatible Service Assignments*

After having discussed the composition test generation and execution in the previous sections, we now evaluate the mechanism for actually detecting faulty and incompatible service assignments in TeCoS. Again, we take the test setup of the event-based composition scenario of Section 6.3 (composition with 3 faulty service combinations along data flow paths of length 2 and 3, respectively). In the following, T denotes the set of composition test cases (same notation as in Section 4.2), which contains 74 test cases in our experiment (64 cases in the minimal set computed by FoCuS, plus 10 additional tests).

Recall from Section 4.2 that a service assignment x is defined as $x : A \to S$, and let $X \in \mathcal{P}([A \to S])$ be the set of all service assignments for which the fault contribution evaluates to 1. More formally, $\forall x \in X : cont(x, T) = 1$. The general idea is that, given the knowledge of succeeded and failed test compositions at any point in time, the service assignments in X which have always led to a fault are deemed incompatible. However, we will see that, as new test results become available, a service assignment which is assumed to be incompatible at some point in time does not necessarily stay in this state.

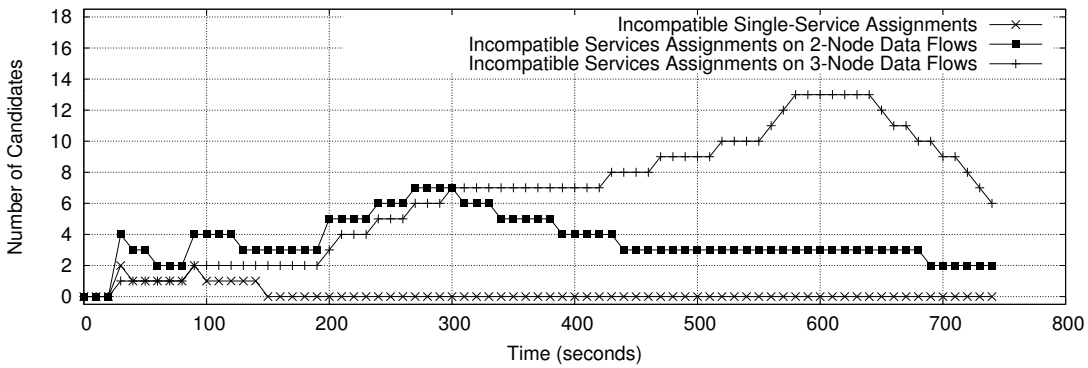

Figure 12. Incompatible Service Assignments Along Data Flows of Different Lengths.
The minimal set (64 tests in this example) ensures that all desired combinations are covered; the 10 additional tests (seconds 650-740) further narrow down the fault localization result.

Figure 13 plots the number of incompatible service assignments over time. The three curves in the figure represent the number of incompatible service bindings along data flow paths of length 1, 2 and 3 (denoted curve I, curve II and curve III). The special case of a 1-node data flow (curve I) actually represents a single-service binding. Up to time point 20, there are no faulty compositions, hence no incompatible service assignments exist. At second 30, the first faulty composition is recorded, and curve I takes value 2, curve II jumps to a value of 3, and curve II reaches value 1. From second 30 onwards, all three curves show a rising trend until they reach a peak value (2 for curve I, 7 for curve II, and 13 for curve III) and from then on start to decrease. Curve I drops to 0 at time 150, which means that after the 15th test case we detect that no single service is solely responsible for the faulty behavior (i.e., the fault has to involve a combination of 2 or more services).

¡¡¡¡¡¡¡ Updated upstream =======

DOI: 10.1002/stvr

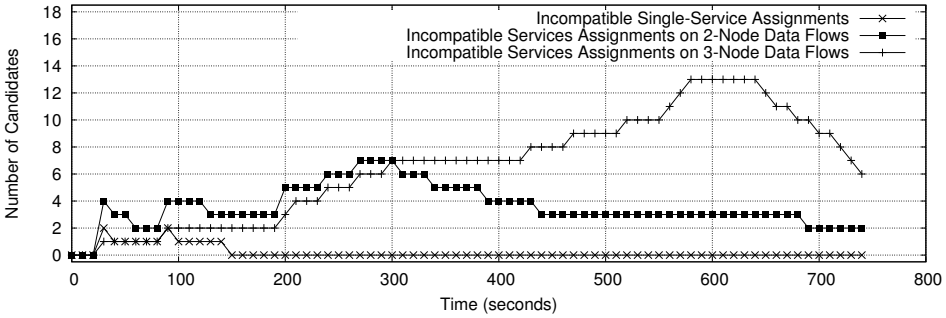

Figure 13. Incompatible Service Assignments Along Data Flows of Different Lengths.
The minimal set (64 tests in this example) ensures that all desired combinations are covered; the 10 additional tests (seconds 650-740) further narrow down the fault localization result.

¿¿¿¿¿¿¿ Stashed changes We observe that the curves in Figure 13 rise and drop over time. The reason for a rise is that a new test case has failed which contains one or more bindings which have not been tested before. Conversely, a drop happens if some binding that previously failed in all test cases is now contained in a successful test case. Overall, it is desirable to narrow down the faulty service combinations as far as possible, i.e., we aim to see the curves at a low level. After executing the minimal set of 64 test cases, at time point 640, curve II has a value of 3, and curve III stands at value 13. We see that the following additional 10 test cases contribute further to the fault localization. This aspect is discussed in Section 6.4.1. Another aspect to consider is the number of incompatible service combinations. In this section we so far considered a composition scenario with 3 injected data flow faults; Section 6.4.2 discusses evaluation results with different faulty data flow combinations.

6.4.1. Additional Tests Beyond the k-Coverage Minimal Test Set In our approach we distinguish between fault detection and fault localization. Generally, the test framework first executes all test cases computed by FoCuS, because they are the minimum requirement to meet the data flow based coverage goal, which ensures that faults are *detected*. To *localize* the origins of faults, the minimal test set may not provide sufficiently precise results (depending on the configuration). The reason for this is as follows. Since the k-node coverage goal attempts to minimize the number of test cases, the longest data flows within a composition are usually represented by only a single test case per possible service combination. That is, within the minimal test set (64 test cases in our experiment) each service combination along the maximum length data flow (3-node data flow in our experiment) is necessarily considered faulty if it is being used in a faulty composition. We can observe this by comparing Figure 11 and Figure 13: up to time point 640, the number of compositions with faulty results is equal to the number of incompatible service assignments on 3-node data flows in Figure 13.

Hence, the tester can decide to include additional tests in the test set T (for illustration, 10 additional tests were run in our experiment). As outlined in Section 4.2, the selection of additional tests is crucial. At this point we generally strive for a reduction of the size of X, i.e., identifying service combinations that *so far* only participated in failed test, but do not necessarily *always* lead to a fault. Therefore, we heuristically construct a new test composition $c \in C$ by picking a (so far deemed to be faulty) service assignment $x \in X$ and filling in concrete services (from previously observed successful compositions) for the remaining abstract services not included in x. By continuously executing such additional test cases, we expect to find a successful composition instance that contains x. This in turn means that x is removed from X, and we are one step further in our quest for the actual origins of faults. For our example, this effect is shown between time points 650 and 740 where ten additional test cases are executed that bring down curve II and curve III to a value of 2 and 6, respectively. While curve-II has now reached its minimum (i.e., the actual expected value), curve-III may be further reduced by executing additional test cases.

DOI: 10.1002/stvr

6.4.2. Coping with Multiple Faulty Service Combinations To illustrate how our approach copes with multiple faulty service combinations, we have run five versions of the experiment mentioned in this section, with increasing number of injected fault combinations (1,2,3,4,5) along (non-overlapping) data flow paths of length 2 and 3.

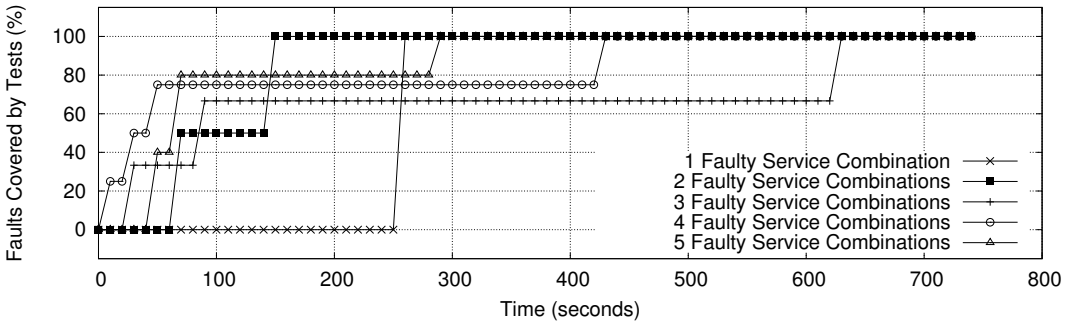

Figure 14. Fault Combination Test Coverage Over Time

Figure 14 illustrates the percentage of faults (i.e., faulty concrete service combinations) that are covered by any of the test cases over time. We can see that the percentage increases as new test cases are executed over time. For instance, the single fault combination is covered after time point 250 and the percentage jumps from 0% to 100%; for 2 faulty service combinations, the first combination is tested at time point 70, and the second fault is covered by the test case at time point 150. The core observation in Figure 14 is that eventually all curves reach 100% on or before time point 640, where the last test case of the minimal test set is executed. This is deterministically reproducible because the k-node coverage criterion ensures that all relevant data flow paths are considered by test cases.

6.5. Discussion of Assumptions, Weaknesses and Limitations

In this section we summarize the main assumptions, weaknesses and current limitations of our approach. This summary also points to open research questions which may be of interest for future investigation.

- Long-running business processes with humans: The focus of ¡¡¡¡¡¡¡ Updated upstream our testing approach is on data-centric, technical Web services and ======= the presented testing approach is on data-centric, technical Web services and ¿¿¿¿¿¿¿ Stashed changes processes with short waiting time between activities. Our approach is arguably not well suited for lengthy service compositions and business processes possibly involving human tasks (e.g., credit application process, product assembly process).
- Testing with real services: The presented framework aims at integration testing of compositions with real (production use) services. The approach is hence best suited for technical services that are free of charge or only associated with a small fee. Integration of expensive services from external providers is not the intended scope. We point out that related approaches which perform *online testing* of services [53, 54, 55, 56] use similar assumptions. For instance, [53] utilizes online testing for service discovery, binding, and composition. Critical security features like authentication and authorization are also best tested in the services' real execution environment [54, 26]. A possible extension to our approach would be to utilize mock testing services which simulate or proxy the actual services. However, some faults and side effects may only be reliably discovered on the basis of the real services.
- Long transitive data dependencies: The k-node data flow coverage criterion is an instrument to limit the size of the test case set. However, the effectiveness of the measure depends on the composition structure. Whereas our approach is well-suited for most real-life data flow graphs (e.g., Composition 1 on the left side of Figure 15), for certain corner cases k-node coverage cannot be applied as effectively (e.g., Composition 2 on the right side of Figure 15). As part of our future work we are further investigating the impact of structural features.

DOI: 10.1002/stvr

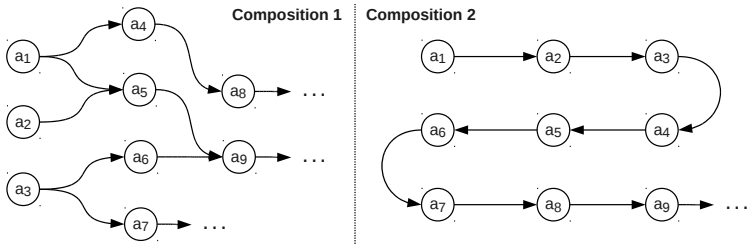

Figure 15. Examplary Data Flow of Structurally Different Service Compositions

- Implicit data dependencies: Moreover, our test model considers only data dependencies on the service composition level. Our approach is not able to detect implicit dependencies in the environment, e.g., caused by two services reading/writing from/to the same distributed file system.
- Lack of semantic information: Currently, our approach lacks semantic information such as knowledge about the internal implementation of services. In particular, a model of the service-internal data flow would help to generate even more accurate test cases. To achieve this task we envision the use of testable services [10, 11] which expose metadata without discovering the actual service internals.

7. RELATED WORK

Testing of SBAs and service compositions has been intensively studied in the previous years. In this section, we discuss some of the related work in these areas in detail.

Earlier works have placed the research areas of service-oriented computing and software testing into perspective, and identified the characteristics and challenges that apply to testing single services and services in combination [12, 57]. An overview and timeline of different testing approaches for service compositions has recently been given by Hazlifah et al. [58]. Our approach is also influenced by early works that introduced data flow oriented program testing (e.g., by Laski/Korel [59] or Rapps/Weyuker [60]), as well as by authors who have proposed data flow analysis as a suitable means to generate test cases for testing Web services (e.g., Heckel and Mariani [61]). Also Liu et al. [62] have addressed the problem of data-flow based testing of Web applications, although their focus is more on analyzing HTML/XML documents and navigation relations across HTTP requests.

In [63], a method to generate test case specifications for BPEL compositions is introduced. Whereas their approach attempts to cover all transitions of (explicit) links between invocations, we focus on direct and indirect data dependencies and ensure k-node data flow coverage.

The data flow-based validation of Web services compositions presented by Bartolini et al [64] has similarities with our approach. The paper names categories of data validation problems (e.g., redundant or lost data) and their relevance for Web service compositions. Data flows in Web service compositions are modeled using Data Flow Diagrams (DFDs). In addition, the authors propose the usage of a data fault model to seed faults (e.g., some value that is out of its domain range) into the data flow model and to establish fault coverage criteria. The DFD can be used either stand-alone to measure structural coverage along data flow paths, or in combination with a BPEL description for checking whether the composition conforms to the data flow requirements. This is contrary to our method: whereas DFDs are defined manually to statically validate the BPEL process, we auto-generate the data flow view and create test cases for dynamic integration testing.

Mei et al. [65] propose a BPEL data flow testing approach which aims at identifying defects in service compositions that are caused by faulty (or ambiguous) XPath expressions selecting a different XML element at runtime than the one that the composition developer intended to be selected. The paper introduces the XPath rewriting graph (XRG) to model XPath on a conceptual level. Based on XRG, different test coverage criteria are defined, which mandate that all possible

DOI: 10.1002/stvr

variants of the XPaths in a BPEL process shall be tested. Other than our work, the paper does not consider dynamic binding or indirect data dependencies in the form of k-node data flows.

In [66], a method for testing orchestrated services is shown. Service orchestrations, e.g., expressed in BPEL, are transformed into an abstracted graph model. Testers define *trap properties* expressed as LTL formulas, which indicate impossible execution traces that should never occur. Model checking is used to generate a counter-examples tree, containing all paths that violate a certain trap property. Testers can further specify which traces are more relevant, which helps pruning the counter-examples tree. This approach is different to ours, as it aims at covering invocation traces of compositions with fixed concrete services. Additionally, it strongly involves the human tester and requires domain knowledge and more manual adjustments in preparation of the test.

Tarhini et al. [67] present an approach for testing Web service based applications, which involves test case generation for several testing steps. Firstly, candidate services are identified based on boundary value testing analysis [68]. Secondly, the candidates are individually tested, making use of a state machine based model of the service internals. Finally, at the integration level, service compositions are tested by covering all possible paths defined in the composition model. A major difference to our work is that services are selected during development time and dynamic binding is not considered.

The authors of [69] present another minimum coverage method for composite services. A description model is introduced, which defines both individual Web services and relationships among these services. Other approaches use Petri nets [70] or extended types of finite state machines [71] to model Web service compositions and to generate tests based on these models.

Testing of dynamic service compositions is related to integration testing in component-based systems [72]. The recent survey in [73] contains a thorough discussion of various issues in integration testing of component based systems, some of which are also closely related to testing service compositions. The work of Piel et al. [74] is largely centered around realizing and testing virtual components in component models. Virtual components enclose a set of real components that are to be tested in combination. Certain structures of virtual components have similarity with service combinations defined by the k-node data flow criterion, but virtual components suffer from problems such as ill-formed empty data flows, which cannot occur in our approach.

The nature of event-based systems and service compositions poses difficult challenges to testing and debugging [75, 76]. For instance, event correlation is necessary to ensure that incoming event messages are associated with the correct processing element, but this mechanism has proven to be complex and often error-prone (e.g., [77, 78]). Additionally, event-based compositions evolve over time and require a means to dynamically initiate new or terminate existing event streams, which is another potential point of failure. Additionally, QoS characteristics are a key concern, particularly when the event processing platform operates under high load. While some of the above mentioned points have previously been tackled in isolation, we utilize TeCoS to test dynamic event-based service compositions end-to-end, as a whole.

Finally, there is a large field of related work in the areas of test result analysis, fault diagnosis [79], and fault localization [39]. Tu et al. [80] discuss fault diagnosis in large graph-based systems and present efficient algorithms for finding potential failure sources from the set of graph nodes that report an alarm. Our approach to analyzing test results is related to their work, as we also consider the dependency graph to determine faulty service assignments that have caused a set of compositions to fail. Software fault localization (e.g., [36, 37, 38, 39]) is a research area whose primary target is to find faults or bugs on the source code level, and the techniques are partly applicable to dynamic service compositions as well. Other seminal work in the area of fault localization has been recently presented by Masri [40]. The technique attempts to identify faulty program statements based on information flow coverage data of historical program executions. In essence, the likelihood of a statement being faulty is determined by contrasting the percentage of failed to passed executions. We build on this approach and, in order to eliminate false positives and false negatives, additionally define the metrics fault contribution and fault participation which are based on precision and recall known from information retrieval [35].

DOI: 10.1002/stvr

8. CONCLUSIONS

In this paper we have discussed the problem of testing dynamic compositions in data-centric service-based systems, and presented a suitable testing mechanism which is implemented as part of the TeCoS framework. We use an abstracted view of compositions that takes into account the data flow occurring between individual service invocations. Based on an illustrative scenario, we presented our model of data-centric compositions and formalized the k-node data flow coverage criterion. The data flow view is suited to abstract from the actual composition technology, and the framework provides a plug-in mechanism to implement test adapters for concrete target platforms. Two such adapters have been implemented and evaluated, one for the de-facto standard service composition language BPEL and the other for a platform for continuous and event-based Web data aggregation. In our previous work we have introduced the notion of k-node data flow coverage for BPEL compositions [16], whereas in this paper we focus on heterogeneous composition environments (BPEL and WS-Aggregation), large-scale test execution, and systematic analysis of test results.

As part of our ongoing work on TeCoS we strive to unify the orthogonal fields of interface-based service testing and service composition testing. So far, our focus has been on dynamic binding of services, and service incompatibilities were identified based on the data dependencies defined in the composition. We are currently extending the approach by analyzing the binding configuration in combination with the input data (see, e.g., [42]). Furthermore, we envision potential future research directions by integrating the concept of testable services into our approach. It will be interesting to see whether different coverage data (e.g., line/branch coverage) exposed by testable services can be utilized to further narrow down the search for faults in dynamic data-centric compositions. Moreover, we are extending the framework to support further service composition techniques, particularly focusing on emerging fields such as data-centric service mashups [45] and interactions in mixed service-oriented systems [81]. We also plan to improve the test generation algorithm and to provide the tester with augmented control over the characteristics of the test optimization and execution. Finally, we are currently integrating the presented testing approach with our recent work on fault modeling in event-based systems [82]. The core idea is to obtain a more complete picture of the processing logic of event-based service compositions (including event correlation, input-output functions, or deployment topologies), in order to perform systematic tests which aim at identifying common sources of faults.

ACKNOWLEDGEMENT

We gratefully thank Eitan Farchi, Rachel Tzoref-Brill, Dan Pelleg and Karen Yorav for their valuable support, conceptual contributions and assistance in the realization of this work. We would also like to express our appreciation to the anonymous reviewers for their detailed feedback, which helped to further improve the paper.

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreements 215483 (S-Cube), 257574 (FITTEST) and 257483 (Indenica). The work was partially supported by the Austrian Science Fund (FWF), grant number P23313-N23 (Audit 4 SOAs).

REFERENCES

1. Papazoglou MP, Traverso P, Dustdar S, Leymann F. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer* 2007; **40**(11):38–45.
2. World Wide Web Consortium (W3C). Web Services Activity. http://www.w3.org/2002/ws/.
3. Dustdar S, Schreiner W. A Survey on Web Services Composition. *International Journal of Web and Grid Services* 2005; **1**(1):1–30.
4. OASIS. Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/OS 2007.
5. Torry Harris Business Solution. SOA Test Methodology. http://thbs.com/pdfs/SOA_Test_Methodology.pdf 2007.
6. TeleManagement Forum. Case study handbook December 2009.
7. World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327/ 2006.
8. Bartolini C, Bertolino A, Marchetti E, Polini A. WS-TAXI: A WSDL-based Testing Tool for Web Services. *2nd IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2009; 326–335.

DOI: 10.1002/stvr

9. Offutt J, Xu W. Generating Test Cases for Web Services Using Data Perturbation. *ACM SIGSOFT Software Engineering Notes* 2004; **29**(5).
10. Eler M, Delamaro M, Maldonado J, Masiero P. Built-in structural testing of web services. *Brazilian Symposium on Software Engineering (SBES)*, 2010; 70–79.
11. Bartolini C, Bertolino A, Elbaum S, Marchetti E. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software* 2011; **84**(4):655–668.
12. Canfora G, Di Penta M. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional* 2006; **8**(2):10–17.
13. Tsai WT, Chen Y, Paul R, Liao N, Huang H. Cooperative and Group Testing in Verification of Dynamic Composite Web Services. *28th International Computer Software and Applications Conference (COMPSAC)*, 2004; 170–173.
14. Tsai WT, Chen Y, Cao Z, Bai X, Huang H, Paul R. Testing Web Services Using Progressive Group Testing. *Content Computing*. Springer, 2004; 314–322.
15. IBM alphaWorks. Focus code and functional coverage tool. http://alphaworks.ibm.com/tech/focus.
16. Hummer W, Raz O, Shehory O, Leitner P, Dustdar S. Test coverage of data-centric dynamic compositions in service-based systems. *4th International Conference on Software Testing, Verification and Validation (ICST)*, 2011.
17. Hummer W, Leitner P, Dustdar S. WS-Aggregation: Distributed Aggregation of Web Services Data. *ACM Symposium On Applied Computing (SAC)*, 2011; 1590–1597.
18. Hummer W, Satzger B, Leitner P, Inzinger C, Dustdar S. Distributed continuous queries over web service event streams. 7th International Conference on Next Generation Web Services Practices (NWeSP) 2011.
19. Hummer W, Leitner P, Satzger B, Dustdar S. Dynamic migration of processing elements for optimized query execution in event-based systems. *1st International Symposium on Secure Virtual Infrastructures (DOA-SVI'11), OnTheMove Federated Conferences*, 2011.
20. Object Management Group (OMG). Business process model and notation (bpmn). http://www.omg.org/spec/BPMN/ 2011.
21. World Wide Web Consortium (W3C). XML Path Language (XPath). http://www.w3.org/TR/xpath/ 1999.
22. Zheng Y, Zhou J, Krause P. Analysis of BPEL Data Dependencies. *33rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2007.
23. Di Penta M, Esposito R, Villani ML, Codato R, Colombo M, Di Nitto E. Ws binder: a framework to enable dynamic binding of composite web services. *International Workshop on Service-Oriented Software Engineering (SOSE)*, ACM, 2006; 74–80.
24. Menascé DA. QoS Issues in Web Services. *IEEE Internet Computing* 2002; **6**(6):72–75.
25. Carminati B, Ferrari E, Hung P. Security conscious web service composition. *International Conference on Web Services (ICWS)*, 2006; 489–496.
26. Hummer W, Gaubatz P, Strembeck M, Zdun U, Dustdar S. An integrated approach for identity and access management in a SOA context. *16th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2011; 21–30.
27. Richardson DJ, Aha SL, O'Malley TO. Specification-based test oracles for reactive systems. *14th International Conference on Software Engineering (ICSE)*, 1992; 105–118.
28. Ramamoorthy C, Ho SB, Chen W. On the automated generation of program test data. *IEEE Transactions on Software Engineering* 1976; **SE-2**(4):293–300.
29. Ammann P, Black P, Majurski W. Using model checking to generate tests from specifications. *2nd International Conference on Formal Engineering Methods*, 1998; 46–54.
30. World Wide Web Consortium (W3C). Web Services Addressing. http://www.w3.org/Submission/WS-Addressing/.
31. World Wide Web Consortium (W3C). Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl 2001.
32. Nie C, Leung H. A survey of combinatorial testing. *ACM Computing Surveys* February 2011; **43**:11:1–11:29.
33. Cohen D, Dalal S, Fredman M, Patton G. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 1997; **23**(7):437 –444.
34. Cheng CT. The test suite generation problem: Optimal instances and their implications. *Discrete Applied Mathematics* 2007; **155**(15):1943–1957.
35. Baeza-Yates R, Berthier RN. *Modern information retrieval*. ACM Press, Addison-Wesley, 1999.
36. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *16th International Conference on Software Engineering (ICSE)*, 1994; 191–200.
37. Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. *20th IEEE/ACM International Conference on Automated software engineering (ASE)*, 2005; 273–282.
38. Renieres M, Reiss S. Fault localization with nearest neighbor queries. *18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003; 30–39.
39. Wong WE, Debroy V. Software fault localization. *Part of the IEEE Reliability Society 2009 Annual Technology Report.* 2009; .
40. Masri W. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability (STVR)* June 2010; **20**:121–147.
41. Inzinger C, Hummer W, Satzger B, Leitner P, Dustdar S. Towards identifying root causes of faults in service-based applications. *31st International Symposium on Reliable Distributed Systems (SRDS)*, 2012; (poster paper).
42. Inzinger C, Hummer W, Satzger B, Leitner P, Dustdar S. Identifying incompatible service implementations using pooled decision trees. *28th ACM Symposium on Applied Computing (SAC), DADS Track*, 2013.
43. Hummer W, Raz O, Dustdar S. Towards Efficient Measuring of Web Services API Coverage. *3rd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS), co-located with ICSE'11*, 2011.
44. Lübke D, Singer L, Salnikow A. Calculating BPEL Test Coverage Through Instrumentation. *ICSE Workshop on Automation of Software Test (AST)*, 2009; 115–122.
45. Benslimane D, Dustdar S, Sheth A. Services Mashups: The New Generation of Web Applications. *IEEE Internet Computing* 2008; **12**(5):13–15.

DOI: 10.1002/stvr

46. Alliance OM. Enterprise Mashup Markup Language (EMML). http://www.openmashup.org/omadocs/v1.0/index.html. URL http://www.openmashup.org/omadocs/v1.0/index.html.
47. Apt KR. *Principles of Constraint Programming*. Cambridge University Press, 2003.
48. Michlmayr A, Rosenberg F, Leitner P, Dustdar S. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Transactions on Services Computing* 2010; .
49. Juszczyk L, Dustdar S. Script-Based Generation of Dynamic Testbeds for SOA. *International Conference on Web Services (ICWS)*, 2010; 195–202.
50. Draheim D, Grundy J, Hosking J, Lutteroth C, Weber G. Realistic Load Testing of Web Applications. *IEEE Conference on Software Maintenance and Reengineering (CSMR)*, 2006; 57–70.
51. Yu T, Zhang Y, Lin KJ. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web* 2007; **1**(1).
52. Leitner P, Hummer W, Dustdar S. Cost-based optimization of service compositions. *IEEE Transactions on Services Computing* 2011; **PP**(99):1.
53. Greiler M, Gross HG, van Deursen A. Evaluation of online testing for services: a case study. *2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS)*, 2010; 36–42.
54. Bertolino A, Angelis GD, Kellomaki S, Polini A. Enhancing service federation trustworthiness through online testing. *IEEE Computer* 2012; **45**(1):66–72.
55. Hielscher J, Kazhamiakin R, Metzger A, Pistore M. A framework for proactive self-adaptation of service-based applications based on online testing. *Towards a Service-Based Internet*, Mähönen P, Pohl K, Priol T (eds.). Springer, 2008; 122–133.
56. Cao TD, Felix P, Castanet R, Berrada I. Online testing framework for web services. *3rd International Conference on Software Testing, Verification and Validation (ICST)*, 2010; 363–372.
57. Bucchiarone A, Melgratti H, Severoni F. Testing service composition. *8th Argentine Symposium on Software Engineering*, 2007.
58. Rusli HM, Ibrahim S, Puteh M. Testing web services composition: A mapping study. *Communications of the IBIMA* 2011; :34–48.
59. Laski JW, Korel B. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering* 1983; **SE-9**(3):347–354.
60. Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* 1985; **11**:367–375.
61. Heckel R, Mariani L. Automatic conformance testing of web services. *Fundamental Approaches to Software Engineering* 2005; :34–48.
62. Liu CH, Kung DC, Hsia P, Hsu CT. Structural testing of web applications. *11th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2000.
63. Garca-fanjul J, Tuya J, Riva CDL. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. *WS-MaTe 2006*, 2006; 83–94.
64. Bartolini C, Bertolino A, Marchetti E, Parissis I. Data Flow-Based Validation of Web Services Compositions: Perspectives and Examples. *Architecting Dependable Systems V* 2008; .
65. Mei L, Chan W, Tse T. Data flow testing of service-oriented workflow applications. *30th International Conference on Software Engineering (ICSE)*, 2008.
66. de Angelis F, Polini A, de Angelis G. A counter-example testing approach for orchestrated services. *3rd IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2010; 373–382.
67. Tarhini A, Fouchal H, Mansour N. A simple approach for testing web service based applications. *International Workshop on Innovative Internet Community Systems*, 2005.
68. Beizer B. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co.: New York, USA, 1990.
69. Zhu Z, Hu Y, Dong X, Li Z. A minimum coverage method for web service composition. *5th IEEE International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2008; 468–472.
70. Dong WL, Yu H, Zhang YB. Testing BPEL-based Web Service Composition Using High-level Petri Nets. *10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, 2006; 441–444.
71. Lallali M, Zaidi F, Cavalli A. Timed modeling of web services composition for automatic testing. *3rd IEEE International Conference on Signal-Image Technologies and Internet-Based Systems (SITIS'07)*, 2007; 417–426.
72. Weyuker E. Testing component-based software: a cautionary tale. *IEEE Software* 1998; **15**(5):54–59.
73. Jaffar-ur Rehman M, Jabeen F, Bertolino A, Polini A. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability (STVR)* June 2007; **17**:95–133.
74. Piel E, Gonzalez-Sanchez A, Gross HG. Built-in data-flow integration testing in large-scale component-based systems. *22nd IFIP International Conference on Testing Software and Systems (ICTSS)*, Springer, 2010; 79–94.
75. Josef Schiefer AS Gerd Saurer. Testing event-driven business processes. *Journal of Computers* 2006; **1**(7):69–80.
76. Beer A, Heindl M. Issues in testing dependable event-based systems at a systems integration company. *2nd IEEE International Conference on Availability, Reliability and Security (ARES)*, 2007; 1093–1100.
77. Barros A, Decker G, Dumas M, Weber F. Correlation patterns in service-oriented architectures. *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, vol. 4422. Springer, 2007; 245–259.
78. Liu G, Mok A, Yang E. Composite events for network event correlation. *Sixth IFIP/IEEE International Symposium on Integrated Network Management*, 1999; 247 –260.
79. Chen J, Patton RJ. *Robust Model-based Fault Diagnosis for Dynamic Systems*. Kluwer Academic Publishers, 1999.
80. Tu F, Pattipati K, Deb S, Malepati V. Computationally efficient algorithms for multiple fault diagnosis in large graph-based systems. *IEEE Transactions on Systems, Man and Cybernetics* 2003; **33**(1):73–85.
81. Schall D, Skopik F, Dustdar S. Expert discovery and interactions in mixed service-oriented systems. *IEEE Transactions on Services Computing* 2011; **99**(PrePrints).
82. Hummer W, Inzinger C, Leitner P, Satzger B, Dustdar S. Deriving a unified fault taxonomy for distributed event-based systems. *6th ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2012.