Test Coverage of Data-Centric Dynamic Compositions in Service-Based Systems

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

[1] Distributed Systems Group
Vienna University of Technology, Austria
{hummer,leitner,dustdar}@infosys.tuwien.ac.at

[2] IBM Haifa Research Lab
Haifa University Campus, Israel
{ornar,onn}@il.ibm.com

Abstract

This paper addresses the problem of integration testing of data-centric dynamic compositions in service-based systems. These compositions define abstract services, which are replaced by invocations to concrete candidate services at runtime. Testing all possible runtime instances of a composition is often unfeasible. We regard data dependencies between services as potential points of failure, and introduce the k-node data flow test coverage metric. Limiting the level of desired coverage helps to significantly reduce the search space of service combinations. We formulate the problem of generating a minimum set of test cases as a combinatorial optimization problem. Based on the formalization we present a mapping of the problem to the data model of FoCuS, a coverage analysis tool developed at IBM. FoCuS can efficiently compute near-optimal solutions, which we then use to automatically generate and execute test instances of the composition. We evaluate our prototype implementation using an illustrative scenario to show the end-to-end practicability of the approach.

1 Introduction

During the last years, the Service-Oriented Architecture (SOA) [16] paradigm has gained considerable importance as a means to create loosely coupled distributed applications. Services, the main building blocks of SOA, constitute atomic, autonomous computing units with a well-defined interface, which encapsulate business logic to provide a certain functionality. Today, Web services [21] are the most commonly used technology to implement service-based systems (SBS) that follow the SOA paradigm. One of the defining characteristics of services is their composability, i.e., the possibility to combine individual services to create a new functionality. The de-facto standard for creating Web service compositions is the Business Process Execution Language for Web Services [14] (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 SBSs allows to define a required service interface at design time and to select a concrete service endpoint from a set of candidate services at runtime.

SBSs require thorough testing, not only of the single participants but particularly of the services in their interplay [18]. Initial testing of a dynamic composite service plays a key role for its reliability and performance at runtime. For one thing, the tests may reveal that a concrete service s cannot be integrated at all 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. For another thing, the test outcome can assist in the service selection process, favor certain well-performing service combinations and avoid configurations for which the tests failed. Furthermore, upfront testing makes it safer and possibly faster to move to a new binding when a new service implementation becomes available in the running system.

However, testing dynamic composite SBSs is a challenging task due to two main reasons. First, white-box testing procedures are only available if the tester has access to the source code, but usually the service implementation is hidden and providers only publish the service interface. In the case of Web services, the interface is provided in the form of a WSDL (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 services. That is, for any nontrivial binding, the number of runtime 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., [3] or [15]. The second problem is con-

sidered one of the main challenges in service testing [6], and has previously been addressed in group testing of services [20, 19]. 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 that approach and provides techniques for restricting the combinations of services and 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-centric coverage goals. 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 contribution is threefold:

- We illustrate and formalize a data-centric view of service compositions in a high-level, abstracted way. We introduce *k-node data flow* coverage, a novel metric expressing to what extent the data dependencies of a dynamic composition are tested. Based on this coverage metric, we formulate the problem of finding a minimum number of test cases for a service composition as a combinatorial optimization problem (COP). Limiting the level of desired coverage helps to significantly reduce the search space of service combinations.
- In order to solve the COP, we make use of FoCuS [8], a tool for coverage analysis and combinatorial test design. We provide an automated transformation between the service composition model and 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 introduce the *TeCoS* (Test Coverage for Service-based systems) framework, which is a prototype implementation of the presented approach. TeCoS stores meta information about SBSs, logs invocation traces, and measures different coverage metrics. This combined information is used to generate and execute service composition test cases. Our evaluation demonstrates the end-to-end practicability of the solution.

1.1 Approach Overview

In this section we briefly describe the solution, end-to-end, at a high level. 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 output generated. The input to the first activity is a composition definition document. This activity converts the input document into a corresponding data flow view, which shows the data-related dependencies between invocations of the composition (see Section 2 for details). In the illustrative scenario implemented in this paper, the composition is defined as a BPEL process, however the abstracted data flow view is generic and can be applied to other composition formats besides BPEL.

The second activity determines the candidate services for each invocation 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 symbolizes the optimization algorithm (executed by FoCuS), which computes a near-optimal solution for the model produced by activity three. In activity five this solution, together with test input/output combinations, is used to generate the actual composition test cases. These are then deployed, executed and summarized in a test report by activity six. 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 six activities are automated and require no human labor.

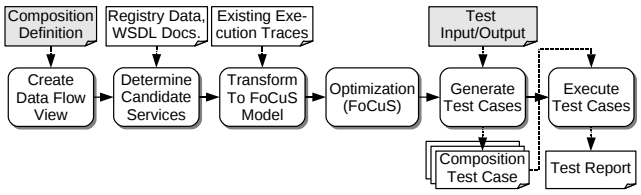

Figure 1. End-to-End Approach

The remainder of this paper is organized as follows. Section 2 presents the concept of dynamic, data-centric compositions based on an illustrative scenario. In Section 3 we discuss the details of the combinatorial test design. Section 4 outlines the functionalities of the TeCoS framework and provides an end-to-end view of the testing approach. Section 5 contains a performance evaluation, and Section 6 points to related work in the area of testing SBSs and service compositions. Section 7 concludes the paper with an outlook for future work.

2 Dynamic, Data-centric Compositions

Recent trends in service computing research and practice show an emphasis on service composition, i.e., combining different services across the Web to create new business functionalities. Testing of service compositions has

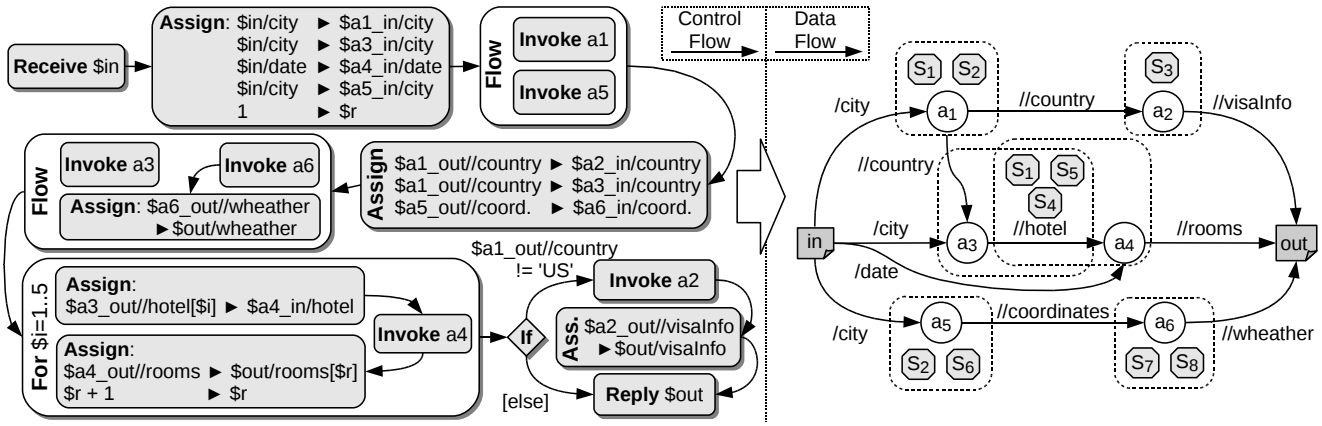

Figure 2. Scenario – BPEL Process View (left) and Data Flow View (right)

become a vivid research area (see related work in Section 6). We are focusing on the particular aspect of data dependencies between services, as these dependencies have a major influence on the overall composition and constitute a potential point of failure. In this section we first introduce a scenario composition to illustrate this aspect. We then formalize the service composition model, and define the test coverage goal that should be achieved.

2.1 Scenario

We base the description of dynamic data-centric compositions on a simple illustrative scenario of a US citizen who plans a trip to a city and requests weather information, available hotel rooms and visa regulations for the target country. This functionality is implemented as a composite Web service using BPEL. Figure 2 depicts the according service composition: the left part of the figure contains a graphical representation of the BPEL process, and the right part illustrates the data flow between the services of the process. While the BPEL process is defined by the composition developer, the data flow view can be generated automatically from the BPEL definition. Note that although this scenario is imaginary and simple, BPEL is often used for complex, large and possibly long-running business processes.

The scenario composition receives an input (`in`) and embraces six abstract services $\{a_1, ..., a_6\}$ to produce the output (`out`). The input contains two XML elements, `city` and `date`. In BPEL, the use (invocation) of 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 information is defined via an XPath [22] expression. 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 city's geographical coordinates, which are used to obtain a weather forecast with service a_6. In the BPEL view, arrows signify the control flow between the activities. The structured `flow` activity signifies that the contained invocations can be executed in parallel. The `for` activity defines a loop that is executed five times to receive only available rooms of the best (first) five hotels. The purpose of the `if` branch is to request the visa information only if the city is not in the US.

The right part of Figure 2 shows the BPEL process definition transformed into the service composition data flow view. The data flow view we use is a simplified version of the BPEL data dependencies model presented in [23]. Whereas they model the actual structure of the BPEL process, our view abstracts the composition process and is agnostic of control flow instruments such as the BPEL `flow` or `for` activities. This view therefore also lends itself to model data flows in non-BPEL service compositions. 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 embraces the concrete services that can deliver the desired functionality; e.g., the abstract service a_1 is implemented by the Web services s_1 and s_2. At runtime, the composition selects and binds to a concrete endpoint, e.g., according to nun-functional and QoS (Quality of Service) parameters. 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). E.g., using s_1 for a_3 and s_4 for a_4 results in an incompatibility and is not allowed. Note that such constraints are defined as additional information and are not directly reflected in the data flow view.

2.1.1 Runtime Composition Instances

Dynamic binding allows replacing each of the abstract services with an invocation of a concrete service at runtime. Table 1 lists the possible combinations of service endpoints 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 instance that is used within a certain combination (row) for a certain 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 1. Possible Combinations of Services

We observe that services with data dependencies (i.e., those that are connected by an arrow) have a direct influence on one another and create a potential point of failure. For instance, say the coordinates in the result of a_5 are encoded as a string, and s_7 can correctly interpret the result from s_2, but fails to parse the coordinates that s_6 delivers. 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 3). 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 1). 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 unfeasible for larger instances.

2.2 Service Composition Model

In the following we define the data-centric service composition model that serves as the basis for the remaining parts of this paper. Table 2 lists and describes the variables that are used in the problem formalization. The left column contains the symbols and variable names. The right column contains a description of the respective symbol and provides an example in reference to the scenario in Section 2.1.

Symbol	Description
$A = \{a_1, ..., a_n\}$	Set of abstract services that are used in the composition definition. Example: $A = \{a_1, ..., a_6\}$
$S = \{s_1, ..., s_m\}$	Set of concrete service endpoints available in the service-based system. Example: $S = \{s_1, ..., s_8\}$
$s : A \rightarrow \mathcal{P}(S)$	Function that returns for an abstract service all concrete service endpoints which provide the required functionality. Example: $s(a_3) = \{s_1, s_4, s_5\}$
$c_x : A \rightarrow S$	Runtime composition instance. The function maps abstract services to concrete services the composition binds to. Example: c_4: $a_1 \mapsto s_2$, $a_2 \mapsto s_3$, $a_3 \mapsto s_4$, $a_4 \mapsto s_4$, $a_5 \mapsto s_2$, $a_6 \mapsto s_7$
$C = \{c_C^1, ..., c_C^j\}$	Set of runtime composition instances. This is the encoding of a solution, i.e., the set of test cases to be executed. The goal is to minimize its size. Example: $C_{min} = \{c_4, c_{12}, c_{13}, c_{17}, c_{20}, c_{21}\}$
$F = \{f_1, ..., f_l\}$, $f_x \in A \times A$	Set of direct data flows (dependencies) between two services. Possible data flows spanning more than two services can be derived from F (see Section 2.3). The function $path(f_x)$ returns the XPath expression(s) associated with data flow f_x. Example: $F = \{(a_1, a_2), (a_1, a_3), (a_3, a_4), (a_5, a_6)\}$
$o(s_x, a_y) \stackrel{\text{def}}{=} \lvert\{c_z \in C : c_z(a_y) = s_x\}\rvert$	Occurrences of concrete service s_x as the endpoint for abstract service a_y in any of the compositions in set C. Example (cf. Table 1): $o(s_2, a_5) = 3$

Table 2. Description of Variables

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 across all compositions c_z in the composition set C.

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

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

We will make use of the equations 1 and 2 later in the paper, to specify that all service instances are to be tested with the same "intensity", i.e., that the difference between $min(a_y)$ and $max(a_y)$ should be minimized.

2.3 k-Node Data Flow Coverage Metric

Before proceeding with details of the combinatorial test design, we take a closer look at data dependencies. Different test coverage metrics for SBSs have been defined. For example, Lübke et al. present activity, branch, link and handler coverage for BPEL processes [10]. Mei et al. take data flows into account [11], but their criteria focus on ambiguous XPath expressions and demand that every possible variant of an XPath should be covered by at least one test case. To the best of our knowledge, no coverage criteria which target the inter-invocation data dependencies in dynamic service compositions have been proposed so far. Such dependencies are important to test because of their potential effect on the correct behavior of the composition.

The data flow view of compositions allows for an analysis of the dependencies between invocations and possible points of failures. As already mentioned in Section 2.1, invocations with direct data dependencies are prone to errors. However, indirect dependencies in a sequence of invocations, e.g., $a_1 \rightarrow a_3 \rightarrow a_4$, should be considered as well. In this example, there are two possible dependencies: 1) if a_3 outputs part of the input it received from a_1, a hidden, but direct dependency between a_1 and a_4 is established; 2) 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 $cvg(d_k)$ of a k-node data flow d_k is a set of concrete service combinations, such that all service combinations of d_k are covered: $\forall s^1_{dk} \in s(a^1_{dk}), ..., s^k_{dk} \in s(a^k_{dk}) : \langle s^1_{dk}, ..., s^k_{dk}\rangle \in cvg(d_k)$.*

Definition 3. *A service-based system 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.*

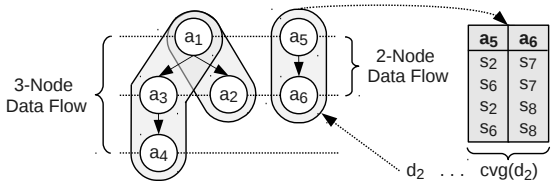

Figure 3. Data Flows in Scenario

Definition 3, stated in other words, asserts that a composition is *k-coverage tested* if all data flow paths of length 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 does not ensure that the composition operates correctly in all situations (even in the case that no data flow involving more than k services exists in the scenario). Actually, this metric is only reasonably applicable if the underlying services have been unit tested with appropriate V&V (Verification & Validation) techniques. Also, k-node data flow coverage does not take into account the different content of the data that is passed between invocations. Hence, the metric's effectiveness 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 services' interface (e.g., extreme values) and the problem domain.

3 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 hard computational problem. The number of possible compositions is exponential in the number of services; instances of the problem of finding minimal sets under constraints in such spaces can be reduced to problems such as the *clique* and *vertex cover* known from graph theory. Hence, at least some instances of the problem are NP-hard. We observe that it is simple to construct any valid solution and to determine the validity of solutions, but no polynomial-time algorithm exists that can guarantee to deliver an optimal solution. The size difference between 1) a (near-)optimal solution and 2) a test set obtained by full enumeration (cf. Table 1) 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.

3.1 Optimization Target

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

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

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 is used for the 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.

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

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:

$$c(a) \neq \emptyset, \forall c \in C, a \in A \qquad (5)$$

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

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

Equation 5 signifies that a composition must assign one endpoint to each abstract service. Equation 6 expresses that each service endpoint that implements one of the abstract services needs to be invoked 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.

3.2 Transformation to FoCuS Data Model

FoCuS implements algorithms for combinatorial test design and optimization of our target function. In the following we describe the mapping from the service composition model to the data model used by FoCuS (see Table 3).

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.

Composite Service 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 3. Mapping of Model Elements

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 incompatible services we add inequality restrictions.

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 an attribute's values and are a possibility to influence the value computation. Expressed in terms of the service composition model, if the candidate services $s(a) = \{s^1_a, s^2_a, .., s^j_a\}$ of an abstract service a have weights $w(s^1_a), w(s^2_a), .., w(s^j_a) \in \mathbb{R}$, then the (ideal) probability that service s^x_a is chosen for testing the abstract service a is $\frac{w(s^x_a)}{\sum_{s^y_a \in s(a)} w(s^y_a)}$. Note that FoCuS gives no guarantee about value distribution, but weights are taken into consideration as part of the 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 4 gives more information about how the traces are collected.

4 The TeCoS Framework

The work of this paper is integrated into the *TeCoS* (*Te*st *Co*verage for *S*ervice-based systems) framework, which is introduced in the following. 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 4 sketches the architecture of TeCoS in the context of

the SBS implementing our scenario. The Web services are provided and hosted by three different service providers. A service integrator defines and publishes the BPEL process on top of the services. 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 SBS, both for the atomic services and for the BPEL process. Our implementation provides an invocation interceptor that can be plugged into the Java Web services framework[1]. The traces and coverage data can be inspected via the Web User Interface. So far, 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. The core component, which orchestrates the testing process is the Test Manager (TM). The TM repetitively invokes the BPEL composition, and each repetition covers one test case. The BPEL process itself is instrumented in such a way that it dynamically retrieves from the TM the Endpoint Reference (EPR) of the services to invoke. 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).

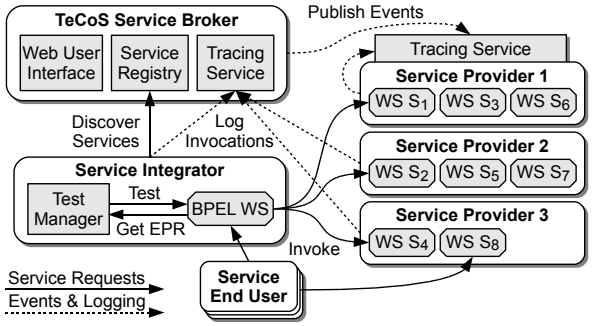

Figure 4. TeCoS Architecture

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 3.2. 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. Similar to the solution proposed in [10], 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. To discuss the details of the single steps that have been outlined earlier in Figure 1, we divide the process into the first three activities (test preparation steps), and the part after FoCuS has obtained the optimized solution (generating & executing BPEL tests).

4.1 Test Preparation Steps

The first step in the test preparation is to create the data flow view from the composition definition, which in our scenario is the BPEL document. To that end, the BPEL `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 we issue a warning message.

```
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 Assignment

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. This solution is tightly coupled to WSDL (services need to implement a certain `portType`), which is not always desirable. Therefore, we also consider the BPEL^{light} [13] approach, which decouples process logic from interface definitions and thereby provides a more flexible means to select service candidates.

The third and last preparatory step is to combine all the gathered information and transform it into the FoCuS data

[1] https://jax-ws.dev.java.net/

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.

4.2 Generating & Executing BPEL 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. This is achieved by instrumenting additional commands into the BPEL definition. The corresponding algorithm is sketched in Algorithm 1.

Algorithm 1 BPEL Instrumentation Algorithm

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

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 6-12), and 2) that the process ID is sent along with the invocation (lines 14-15).

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 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 Evaluation

In this section we evaluate different aspects of the proposed solution. All performance tests were run on a machine with Quad Core 2.8GHz CPU, 3GB memory, running Ubuntu Linux 9.10 (kernel version 2.6.31-22).

First, we investigate how the k-node data flow coverage criterion affects the solution size. Consider three imaginative composition scenarios (S=small/M=medium/L=large), see Table 4. S/M/L have 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. $\|C\|$ for $k=4$	125		10000		160000	
Min. $\|C\|$ for $k=3$	125		1000		8000	
Min. $\|C\|$ for $k=2$	25		100		400	
Min. $\|C\|$ for $k=1$	5		10		20	
FoCuS Results	min	max	min	max	min	max
Execution Time [s]	0.15	0.48	2.91	3.84	742.9	939.3
Test Cases ($\|C\|$)	125	125	10K	10K	160003	160008
Occurrence Diff.	35	48	557	623	5148	5935

Table 4. Optimization of Different Model Sizes

The lower bound of generated test cases, minimum $|C|$, varies with the value of k. For the special case that $k = 1$, minimum $|C|$ equals the maximum number of concrete services per one abstract service, $max(|s(a_x)|), a_x \in A$. In general, minimum $|C|$ equals $max(|s(a_x)|)^k$, provided the composition contains any data flow of length k. Note that these bounds are only valid if 1) the composition is free of constraints (i.e., some abstract services must or must not bind to the same concrete service), and 2) no previous execution traces exist. We applied the FoCuS optimization to the three scenarios (see Table 4). The optimization has been repeated 10 times and the table lists minimum and maximum of execution time (seconds), number of generated test cases and the total service occurrence differences ($\sum_{a \in A}(max(a) - min(a))$). $|C|$ of the FoCuS result is optimal for S and M, and close to minimum $|C|$ for L.

Figure 5 illustrates the upper and lower bound of $|C|$ 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.

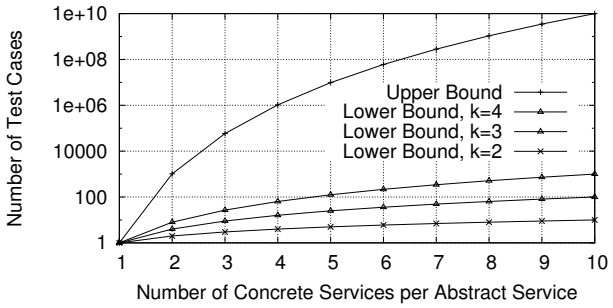

Figure 5. Combinations in Medium Scenario

To evaluate the end-to-end performance, we have implemented and tested the scenario presented in Section 2.1. The scenario BPEL process is deployed in a *Glassfish*[2] server. To simulate the services, we make use of *Genesis* [9], a test bed generator for Web services.

Table 5 lists the execution time of the single steps in the end-to-end testing lifecycle (all values are in milliseconds). Firstly, creating the data flow view from the BPEL definition file took 147ms. This duration depends mainly on the number of `Assign` activities 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. The FoCuS algorithm terminated after 644ms. The largest part of the preparation is the BPEL instrumentation and deployment (around 10 seconds).

Test Preparation		#	Set EPR	Process	Total
Data Flow View	147	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
Deploy BPEL	10275	c_{20}	230.0	386.0	616.0
		c_{21}	320.0	303.0	623.0
Sum	12284	Avg	288.3	384.2	672.5

Table 5. Performance of Test Scenario

The remaining values in the table relate to the six generated test cases. 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 these activities. The timestamps are sent to the Test Manager Service at the end of the process execution to calculate the total duration. On average, this duration was 288.3ms, which involves setting all dynamic EPRs according to the test case. The execution time of the actual business logic depends on the domain, in our case it was around 384.2ms, totaling in an average time of 672.5ms per test case.

[2]https://glassfish.dev.java.net/

6 Related Work

In the following we discuss related work in the area of testing SBSs and service compositions.

In [7], 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 [2] 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.

The BPEL data flow testing approach in [11] 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 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 [1], 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 service endpoints. Also, it strongly involves the tester and requires domain knowledge and more manual adjustments in preparation of the test.

[17] presents 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 [4]. Secondly, the candidates are individually tested, making use of a state machine based model of the services' internals. Finally, at the integration level, service compositions are tested by covering all possible paths defined in the composition model. The main difference to our work is that services are selected during development time and dynamic binding is not considered.

7 Conclusions

In this paper we have discussed the problem of testing dynamic compositions in service-based systems. 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. We presented our end-to-end implementation based on BPEL, the de facto standard for Web service composition. We pointed out that the presented model also supports other composition techniques. Our implementation is integrated into TeCoS, a framework that stores service metadata and logs invocation traces in order to generate and execute composition test cases.

As part of our ongoing work we are extending the framework to support alternative service composition techniques, particularly focusing on the emerging field of data-centric service mashups [5], formats like the Enterprise Mashup Markup Language[3] (EMML), and frameworks such as Yahoo pipes[4]. We also strive to improve the test generation algorithm, and to let the tester control the characteristics (e.g., maximum execution time) of the optimization. As a replacement for the current service registry, we envision the integration of a more sophisticated framework such as VRESCo [12], which provides QoS metadata, complex event processing and more. Furthermore, we plan to parallelize the test case execution on multiple server instances.

Acknowledgements

We gratefully thank Eitan Farchi, Rachel Tzoref-Brill and Karen Yorav for their valuable support, conceptual contributions and assistance in the realization of this work.

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) and 257574 (FITTEST).

[3] http://www.openmashup.org/omadocs/v1.0/

[4] http://pipes.yahoo.com

References

[1] F. d. Angelis, A. Polini, and G. d. Angelis. A counter-example testing approach for orchestrated services. In *ICST 2010*, pages 373–382, 2010.

[2] C. Bartolini, A. Bertolino, E. Marchetti, and I. Parissis. Data Flow-Based Validation of Web Services Compositions: Perspectives and Examples. *Arch. Depend. Syst. V*, 2008.

[3] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *ICST 2009*, pages 326–335, 2009.

[4] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, USA, 1990.

[5] D. Benslimane, S. Dustdar, and A. Sheth. Services Mashups: The New Generation of Web Applications. *IEEE Internet Computing*, 12(5):13–15, 2008.

[6] G. Canfora and M. Di Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*, 8(2):10–17, 2006.

[7] J. Garca-fanjul, J. Tuya, and C. D. L. Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In *WS-MaTe 2006*, pages 83–94, 2006.

[8] IBM alphaWorks. Focus code and functional coverage tool. http://alphaworks.ibm.com/tech/focus. Accessed 2010-08.

[9] L. Juszczyk and S. Dustdar. Script-Based Generation of Dynamic Testbeds for SOA. In *ICWS*, pages 195–202, 2010.

[10] D. Lübke, L. Singer, and A. Salnikow. Calculating BPEL Test Coverage Through Instrumentation. In *Int. Workshop on Automation of Software Test*, pages 115–122, 2009.

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

[12] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Tr. on S.C.*, 2010.

[13] J. Nitzsche, T. Van Lessen, D. Karastoyanova, and F. Leymann. BPEL^{light}. In *BPM 2007*, pages 214–229, 2007.

[14] OASIS. Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/OS, 2007.

[15] J. Offutt and W. Xu. Generating Test Cases for Web Services Using Data Perturbation. *Softw. Eng. Notes*, 29(5), 2004.

[16] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007.

[17] A. Tarhini, H. Fouchal, and N. Mansour. A simple approach for testing web service based applications. In *Int. Workshop on Innovative Internet Community Systems*, 2005.

[18] Torry Harris Business Solution Inc. SOA Test Methodology. http://thbs.com/pdfs/SOA_Test_Methodology.pdf, 2007.

[19] W.-T. Tsai, Y. Chen, Z. Cao, X. Bai, H. Huang, and R. Paul. Testing Web Services Using Progressive Group Testing. In *Content Computing*, pages 314–322. 2004.

[20] W. T. Tsai, Y. Chen, R. Paul, N. Liao, and H. Huang. Cooperative and Group Testing in Verification of Dynamic Composite Web Services. In *COMPSAC*, pages 170–173, 2004.

[21] World Wide Web Consortium (W3C). Web Services Activity. http://www.w3.org/2002/ws/.

[22] World Wide Web Consortium (W3C). XML Path Language (XPath). http://www.w3.org/TR/xpath/, 1999.

[23] Y. Zheng, J. Zhou, and P. Krause. Analysis of BPEL Data Dependencies. In *EUROMICRO SEAA 2007*, 2007.