Recommending Service Repairs

Joshua Church and Amihai Motro
Department of Computer Science
George Mason University
Fairfax, VA
{jchurch2, ami}@gmu.edu

***Abstract*—We address the issue of failure in service compositions. Such failures occur when a service in the composition evolves or becomes unavailable. Our goal is to analyze these failures and recommend possible service repairs. We begin with a formal model of services and service compositions. In this model, services are abstracted as functions that map input domains to output domains and knowledge of their semantics is restricted to a relatively short table of input-output pairs. Service compositions are then formed with the help of two functional operators. We then design a technique to support programmers in debugging service failure by offering a ranked list of suggested repairs. The repairs may substitute the failed component service or any portion of the composition that contains it. Finally, we describe experiments that validate our technique.**

I. Introduction

After a decade of development, there is a vast amount of information made available via *RESTful* web services, i.e., software that is accessible over a network and is memoryless, stateless, and free of side-effects. For instance, given a movie title, a service may return a list of theater locations, or, given a gene's name, a service may return a description of its function. Additionally, as in scientific computing or e-commerce, new information may be computed in *compositions* that pass returned values through a series of *connected* services found in a repository [9], [22].

The downside of this service-oriented approach is that, after deployment, services within compositions may become inoperable. Such failures occur when a service becomes unavailable, or when its functionality unexpectedly evolves [20]. In response, programmers must repair the composition to recover its intended functionality (at least to a similar degree). This recovery process is generally a manual task: Locate the failure and then substitute it with a service of similar functionality. While techniques for "fault localization" (identify where the code is broken) are available [19], [8], identifying repair candidates is an open problem [18]. Indeed, software maintenance is a well-researched field, but the maintenance of service-oriented software deserves more attention.

We argue that there is pressing need for techniques that support programmers in the debugging of service failures. In this work, our goal is to analyze the options and recommend a list of service repairs, ranked in order of similarity to the original service. Whereas most automated repair proposals assume knowledge about program source code [21], [15], we consider services to be "black-boxes"; that is, implementation details such as source code or documentation are unavailable.

Our model abstracts services as *functions* that map input values to output values. The *extensions* of these functions, which we call *behavior tables*, provide the semantics of these black-boxes. As behavior tables could be very large (even infinite), we assume that only *samples* of the behavior tables are available. These samples are obtained by querying the services with random values selected from their input domains. In an earlier paper [5] we showed that reliable behavior can be learned from relatively small samples; often with as few as 10 to 32 behavior instances. Given this small size, we could assume that this sample behavior is stored alongside the service in a repository. To express compositions of repository services, we define a simple language that creates service expressions with the help of two operations, called *compose* and *combine*.

Assume now that a repository service fails. In an earlier work [7] we described how this service can be substituted optimally with an alternative chosen from a repository. The new service must be input-output compatible with the failed service, and it should exhibit the closest behavior to it (i.e., maximize a service similarity measure that we adopt).

In this work we consider the more complex situation in which the failed service is embedded in a composition. The new situation presents several challenges: First, we must consider the option that, instead of substituting the failed service with another repository service, we might get better results by substituting an entire *sub-expression* in which the failed service is embedded (possibly even the whole composition). The new composition could be more efficient in that it would achieve similar functionality with fewer services. This requires the repair procedure to consider all the sub-expressions in which a failed service is embedded.

Furthermore, choosing the "best" substitution cannot be done "locally", where the failed service (or a sub-expression that contains it) is compared with the new alternative: We must compare the behavior of the original composition to the behavior of the new composition that results. Obviously, in these processes we must have access to the behavior of compositions. The challenge here is that, unlike repository services, the behavior of compositions is not readily available, and must be *inferred* from the behavior samples of its component services.

Our main contribution is a procedure that considers all the possible substitutions and identifies the one that results in the optimal repair. Pragmatically, our method applies this

978-0-7695-5306-1/14 $31.00 © 2014 IEEE
DOI 10.1109/MESOCA.2014.10

procedure to derive a *ranked list* of repairs that are recommended to the programmer. This provides the programmer the opportunity to apply his judgment. Obviously, there could be situations where no recommended repair is deemed acceptable since a service "similar enough" to the failed service could not be found.

Several aspects of our method were validated, the most significant being whether the suggested repairs are of high quality. Our experiment showed that when the list of suggested repairs is of length 10, in 85% of the times it suggested a repair that had the same semantics as the failed composition.

Our model is described in Section III, where we formally define services and a language for their composition. Section IV describes our technique for recommending service repairs. The details and analysis of our experiments are provided in Section V. Finally, Section VI summarizes the results and outlines future research directions. We begin in Section II with a brief survey of related work.

II. Background

Our work focuses on the recommendation of service repairs after a component service fails. Hence, we introduce a model and technique to maintain functionality of deployed service compositions. It is, therefore, helpful to consider prior research in three active software maintenance topics: adaptive software, code search, and automated software repair.

Correction after unexpected software modification is also a theme in adaptive software research [1]. This field explores techniques to adapt software in response to changes to user needs, to run-time environments, or to program properties. In the context of service-oriented software, the goal is to maintain service functionality by reconfiguring its architecture (rearrange sequences of services such that different service compositions perform the same functional requirements), by adjusting its parameters (refine the properties of service in a composition), or exchanging its components (replace a single service in compositions with an equivalent service [2]). We call the third type of adaptation a substitution and it is the emphasis of our review.

Given a multitude of services that perform similar functions, this raises the issue of choosing the "best" substitute. There are numerous approaches that define the concept of best, but generally, the objective is to select the *most relevant* services from among a set of choices. In [26], [24], [27], differences in quality of service (QoS) parameters are considered when choosing between equivalent services; in contradistinction, we emphasize differences in observable behavior.

Additionally, these works assume information about service functionality is given by external sources whereas we learn service functionality from run-time traces [6]. For this purpose, our approach is facilitated by instrumenting software as it runs and then collecting invocation traces in a repository [13]. Of course, this type of information can also be gleaned from deployment logs or other histories of program executions [14], [11].

Given a query service, our task is to find similar services from a repository such that one may replace the other. Towards this end, many approaches to code search adapt information retrieval techniques (e.g., classification schemes, free-text indexing, and relational databases) to indexing software artifacts [12], [25], [3], [10]. These works assume textual descriptions of functionality are given. However services are generally offered without source code or documentation, and available WSDL files that describe service functionality are often incomplete, inaccurate, or ambiguous. For this reason, we use outputs from a system-supplied sample of inputs.

Furthermore, these works retrieve services based only on exact keyword matches whereas we define a general method to measure the similarity between services [7]. Thus, an essential concept within our model is service similarity, and an essential issue is the measurement of two compatible service behaviors. This is an example of the widely researched nearest neighbor problem in metric spaces [23], [28] where "nearest to a given query service" is interpreted as the service with the smallest distance to an example service.

A second essential issue we address is the formal representation of service compositions. Similar to our approach, the research in [17] uses input-output behavior to model program behavior; however, it does not consider intermediate program states such that occur in service compositions. In addition, the research of [16] uses input-output behavior to compose loop-free programs built from a repository, but in contrast, its formal representation of semantics is captured in constraints derived from output values whereas our representation of semantics is captured in tables of output values.

Finally, the work in [4] proposes a technique to find sequences of services that act as a 'workaround' for a failed component service. Like our research [4] supports repair of service-oriented programs; however, their technique requires knowledge as to which services are substitutable. On the other hand, our work uses services similarity to find service substitutes.

III. A Model of Services and Service Compositions

A good substitute for a given service should be input-output compatible and should behave similarly. To this end, our model defines services so that their syntax and semantics can be captured accurately. We begin by defining the syntax and semantics of *services*. We then define a formal language for service compositions, and associate semantics with compositions as well.

A. Services

Before defining services, we formalize the notion of *domains*. A domain is a set of values that share similar syntax and semantics. A domain is either *simple* or *complex*. A simple domain is a set of scalar values of a certain *type*. For now, we assume only two types: numbers and character strings. The values of a domain have shared semantics. For example, a domain may be a range of temperatures, a set of Zip codes, or a set of stock symbols.

Simple domains may be combined to form complex domains with two operators, *aggregate* and *sequence*. Given a domain α, the *aggregation* of α forms a new domain, denoted $\{\alpha\}$, in which each element is a *set* of elements of α. Given domains $\alpha_1, ..., \alpha_n$, the *sequencing* of $\alpha_1, ..., \alpha_n$, forms a new domain, denoted $(\alpha_1, \ldots, \alpha_n)$, in which each element is a sequence of n elements in which the i^{th} element is from the domain α_i. For example, a set of Zip codes may be aggregated to denote a destination. As another example, values of latitude, longitude and elevation may be sequenced to denote a 3-position coordinate.[1]

These two operators may be applied repeatedly to create arbitrarily complex domains from simple domains. Let α denote a simple domain, then the grammar for defining domains Δ is: $\Delta ::= \alpha | \{\Delta\} | (\Delta, ..., \Delta)$.

We view services as software components that provide data in response to requests. These components are stateless and free of side-effects; that is, their responses depend only on the input in the requests, and these responses are the only outcome of the requests. Hence, our model abstracts information services as functions from input domains to output domains: Let A and B be two domains, a service s is a function $s : A \longrightarrow B$.

An example of a service over simple domains is the service s_1 that receives the name of a country and returns its capital city $s_1 : country \longrightarrow city$. Examples of services over complex domains are the service s_2 that receives a set of Zip codes and returns the current average temperature and humidity $s_2 : \{zip\} \longrightarrow (temp, humidity)$; the service s_3 that receives the title of a movie and return pairs of theaters and show times $s_3 : title \longrightarrow \{(theater, time)\}$; and the service s_4 that receives a combination of stock symbol and date and returns the opening, closing, low and high prices for that date $s_4 : (stock, date) \longrightarrow (open, close, high, low)$.

Finally, two services s_1 and s_2 are said to be *compatible* if they have the same input domain and the same output domain. A repository of services $\mathcal{R}$ may thus be *partitioned* into subsets of compatible services. It should be emphasized that this partitioning is largely syntactical, and compatible services could behave very differently. For example, there could be several compatible services from the input domain *stock* to the output domain *price*. However, one would return the most recent closing price of the stock, another could return the 90-day average, and yet another could return the 5-week high. A good substitution for a given service should not only be compatible, but should also *behave* similarly. The behavior of a service is provided by the extension of the function:

The *behavior* of a service $s : A \longrightarrow B$ is the set of matched pairs $S = \{(a, s(a)) \mid a \in A\}$. An individual pair $(a, s(a))$ is an *instance* of the service s.

Behavior may be represented in a two column *table*, in which each row is an instance. For example, assuming the input domain has cardinality p, the behavior of s has p instances:

S:

A	B
a_1	b_1
$\vdots$	$\vdots$
a_p	b_p

It is often useful to view the service s as a simple query processor of a single table that can handle one type of query: given a value $a \in A$ it retrieves $\pi_B(\sigma_{(A=a)}(S))$.

Our definitions of service and service behavior require two pragmatic adjustments that will facilitate practical methods for repairing failed services. The first adjustment considers the issue of very large (or even infinite) input domains that result in intractable behavior tables. We address this issue by considering only *samples* of behavior tables. Such samples must be statistically valid; namely, random and of sufficient size. Of course, by using samples rather than the "true" tables, our methods work with *estimated* behavior. A sample (estimate) of behavior table S will be denoted $\overline{S}$. An efficient method for obtaining satisfactory samples was described in [5].

Formally, a service should respond to inputs that are in its domain, whereas in real-world situations, services may be invoked with inputs that are outside their domains. Hence, our second pragmatic adjustment introduces a special value called *exception* which is added to every domain. When a service receives an input outside its domain (or when it receives *exception* input), it returns *exception* in each of its outputs. This implies that tables of estimated behavior may contain exceptions.[2] Consider this small 5-instance sample in which service exceptions are denoted with —:

A	B
a_1	b_1
a_2	—
a_3	b_3
a_4	b_4
—	—

B. Service Compositions

Recall that our goal is to develop methods for repairing compositions that stop functioning due to the failure of a component service. Toward this goal, we formalize the concept of service compositions. Compositions are formed from individual services (which we call *basic* services), with two operators, *compose* and *combine*. The expressions that result define new *composite* services that model the flow of data among a set of basic services retrieved from the repository. The *compose* and *combine* operators are both binary: each connects two "compatible" services to create a composite service. The *compose* operator executes services in *sequence*, whereas the *combine* operator executes services in *parallel*. Together, they allow creating complex new services.

[1] Note that in set-theoretical terms, the sequence of domains corresponds to their Cartesian product.

[2] These *exception* values are similar to database *null* values.

Consider two services:

$$\begin{aligned} s_1 : A &\longrightarrow B \\ s_2 : B &\longrightarrow C \end{aligned}$$

Note that the co-domain of s_1 is the domain of s_2. The *compose*, denoted $s_1 \circ s_2$, is defined as the composition of the two functions:

$$\begin{aligned} s_1 \circ s_2 : A &\longrightarrow C \\ (s_1 \circ s_2)(a) &= s_2(s_1(a)) \end{aligned} \tag{1}$$

It is often desirable to connect the outputs of two (or more) services to the inputs of another service, or the output of one service to the inputs of two (or more) services. Towards this goal, we define the *combine* operator. Assume

$$\begin{aligned} s_3 : A &\longrightarrow C \\ s_4 : B &\longrightarrow D \end{aligned}$$

The *combine* operator, denoted (s_1, s_2), is defined as the product of the two functions:[3]

$$\begin{aligned} (s_3, s_4) : (A, B) &\longrightarrow (C, D) \\ (s_3, s_4)(a, b) &= (s_3(a), s_4(b)) \end{aligned} \tag{2}$$

To demonstrate the use of both operators together, assume another service s_5:

$$s_5 : (C, D) \longrightarrow E$$

We compose (s_3, s_4) with s_5 to create $(s_3, s_4) \circ s_5$:

$$\begin{aligned} ((s_3, s_4) \circ s_5) : (A, B) &\longrightarrow E \\ ((s_3, s_4) \circ s_5)(a, b) &= s_5(s_3(a), s_4(b)) \end{aligned} \tag{3}$$

The composite service $(s_3, s_4) \circ s_5$ connects the outputs s_3 and s_4 to the input of s_5.

In duality, we can use *combine* and *compose* to connect the outputs of one service to the inputs of two (or more) services connect the outputs of one service to the inputs of two (or more) services. Assume services s_6, s_7 and s_8:

$$\begin{aligned} s_6 : F &\longrightarrow A \\ s_7 : G &\longrightarrow B \\ s_8 : E &\longrightarrow (F, G) \end{aligned}$$

We compose s_8 with (s_6, s_7) to create $s_8 \circ (s_6, s_7)$:

$$\begin{aligned} (s_8 \circ (s_6, s_7)) : E &\longrightarrow (A, B) \\ (s_8 \circ (s_6, s_7))(e) &= (s_6(s_8(e)), s_7(s_8(e))) \end{aligned} \tag{4}$$

Finally, we can compose the service described in Equation 4 with the service described in Equation 3 to create the service $(s_8 \circ (s_6, s_7)) \circ ((s_3, s_4) \circ s_5)$:

$$\begin{aligned} (s_8 \circ (s_6, s_7)) \circ ((s_3, s_4) \circ s_5) : E &\longrightarrow E \\ (s_8 \circ (s_6, s_7)) \circ ((s_3, s_4) \circ s_5)(e) &= \\ s_5(s_3(s_6(s_8(e))), s_4(s_7(s_8(e)))) & \end{aligned} \tag{5}$$

[3]We use (s_3, s_4) instead of the more common functional notation $s_3 \times s_4$, and we use (C, D) instead of the more common set notation $C \times D$.

Figure 1 illustrates the compositions described in Equations 1–5. In these diagrams each service is represented with a rectangle with in-coming and out-going arrows. Note that each such arrow represents multiple inputs and outputs (the arrows may be labeled with the inputs and outputs).

The behavior (extension) of service compositions is derived from the behavior of basic services, as implied by Equations (1) and (2). Consider two services s_1 and s_2 with corresponding behavior tables S_1 and S_2. The behavior table of their composition $s_1 \circ s_2$ (Equation (1)) is simply the relational algebra join followed by a projection that removes the join columns $\pi_{A,C}(S_1 \bowtie S_2)$, and the behavior table of their combination (Equation (2)) is simply the Cartesian product $S_1 \times S_2$.

However, recall our pragmatic adjustment to use only samples (estimates) of the extensions of services. This suggests that we must derive the samples of the behavior of compositions from the samples of the behavior of the basic services; that is, from $\overline{S_1}$ and $\overline{S_2}$:

$$\overline{\pi_{A,C}(S_1 \bowtie S_2)} \approx \pi_{A,C}(\overline{S_1} \bowtie \overline{S_2}) \tag{6}$$

$$\overline{S_1 \times S_2} \approx \overline{S_1} \times \overline{S_2} \tag{7}$$

The join in Equation (6) presents a pragmatic difficulty in that some of the values in the output column of $\overline{S_1}$ may not be available in the input column of $\overline{S_2}$ resulting in samples with an insufficient number of instances. This could be remedied by invoking s_2 with the unmatched values in the output column of $\overline{S_2}$. These unmatched values are preserved with the relational algebra left outer join.

As an example, consider services

$$\begin{aligned} s_1 : A &\longrightarrow B \\ s_2 : C &\longrightarrow D \\ s_3 : (B, D) &\longrightarrow E \end{aligned}$$

with the following behavior samples[4]

$\overline{S_1}$

A	B
1	2
2	3

$\overline{S_2}$

C	D
1	2
2	4
3	6
4	8

$\overline{S_3}$

B	D	E
2	2	4
2	4	6
2	6	8
3	2	5
3	4	7
3	6	9

Consider now the service composition

$$(s_1, s_2) \circ s_3 : (A, C) \longrightarrow E$$

The behavior sample for (s_1, s_2) is[5]

[4]Intuitively, s_1 increments its input, s_2 doubles its input, and s_3 sums its two inputs.

[5]Note that the columns have been rearranged.

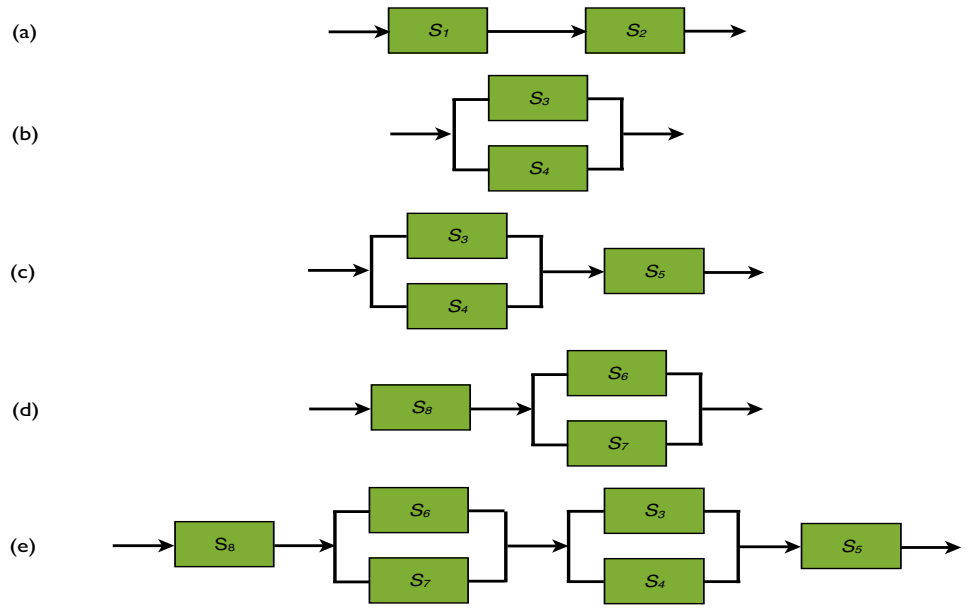


Fig. 1. The *compose* and *combine* operators.

$\overline{(S_1 \times S_2)}$

A	C	B	D
1	1	2	2
1	2	2	4
1	3	2	6
1	4	2	8
2	1	3	2
2	2	3	4
2	3	3	6
2	4	3	8

and for the entire expression $(s_1, s_2) \circ s_3$

$\overline{\pi_{A,C,E}((S_1 \times S_2) \bowtie S_3)}$

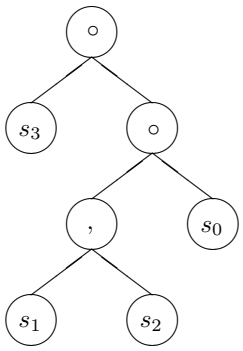
A	C	E
1	2	4
1	2	6
1	3	8
1	4	–
2	1	5
2	2	7
2	3	9
2	4	–

Note that the outer left join of (A, C, B, D) and (B, D, E) introduces two exceptions (null values). These will require invoking s_3 twice more, with the inputs (2,8) and (3,8), to obtain the values 10 and 11, respectively.

IV. Repairing Failed Services

Having described a formal model for services and service compositions, we now turn our attention to the subject of this work, namely the repair of failed services. Our general premise is that a basic service has failed during the execution of a composite service, and that this service must be isolated and replaced so that the composition may be re-executed successfully. More formally, assume a composite service s has failed, and a basic service s_0 which is a component of s has been isolated as the reason for the failure. The task is to configure a new composite service s^* that will be an "optimal" substitution for s (obviously, s^* will no longer engage s_0, using other available services instead).

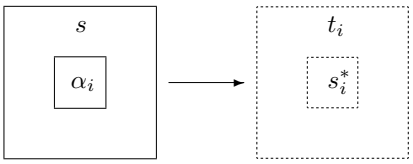
To describe our method, it is beneficial to view composite services as *trees*, in which basic basic services are leaf nodes and the *compose* and *combine* operators are internal nodes. For example, the composite service $s = s_3 \circ (s_1, s_2) \circ s_0$ is represented with the tree



When a basic service (a leaf of the tree) fails, we consider not only replacing this service, but also every sub-expression (sub-tree) that incorporates this service. This reflects the fact

that there might be a service available in the repository that can substitute effectively for an entire sub-expression. As an example, assume the failed service calculates the Zip code for a given city, but, in the given composite service, this service is composed with a service that provides the temperature for a given Zip code. It might be more beneficial to substitute the composition of both with a service that provides the temperature for a given city.

Let s be composite service that incorporates a failed basic service s_0, and let $\alpha_1, \ldots, \alpha_n$ be the sub-expressions of s that incorporate s_0. In particular, α_1 is simply the failed service s_0 and α_n is the entire composite service s. We transform s into a new composite service t_i by substituting a sub-expression α_i with a new service s_i^* which is compatible with α_i and which behaves "most similarly". Note that whereas α_i is possibly a composite service, s_i^* is a basic repository service. It should be intuitively clear that there is no advantage in considering sub-expressions that do not incorporate the failed service. We now have a set $t_1, \ldots, t_n$ of "most similar" reconfigurations of the original composite service s, and we must choose the"most similar" among them.



We have used the intuitive term "most similar service", and now we formalize the concept of service similarity. Consider two *compatible* services $s_1 : A \longrightarrow B$ and $s_2 : A \longrightarrow B$. Assume a metric d is defined that measures the distance between every two elements of B.[6] The similarity between two elements of B is defined as the reciprocal of the distance: $sim(b_1, b_2) = 1/d(b_1, b_2)$. The *similarity* between the two services is defined as the average similarity of their outputs for all possible inputs:

$$sim(s_1, s_2) = \frac{1}{|A|} \sum_{a \in A} sim(s_1(a), s_2(a))$$

When only *samples* of the behavior are available, the average is on the instances of the behavior samples (note that both samples should use the same values of A). For more details, see [7], where this formalization is shown to reflect the inherent similarity among services. When B is a *complex* domain, the metric d is defined recursively from distances that are defined on the simple domains that are involved in the definition of B [7]. Our approach to repairing failed composite services can now formalized in this procedure.

Input: A composite service s that incorporates a failed basic service s_0.

Output: A composite service t that optimally substitutes for s and does not incorporate s_0.

[6]For example if B is a domain of numbers, d could be the absolute value of their difference.

Step 1: Extract the sub-expressions $\alpha_1, \ldots, \alpha_n$ of s that incorporate s_0.

Step 2: For each sub-expression α_i $(i = 1, \ldots, n)$:

1) Construct its estimated behavior table from the behavior tables of the basic services.[7]
2) For each repository service s_i that is compatible with α_i calculate the behavior similarity $\Phi(\alpha_i, s_i)$.[8]
3) Let s_i^* denote the repository service s_i for which the behavior similarity is highest.
4) Replace the sub-expression α_i with s_i^*. Let t_i denote the result.

Step 3: For each composite service t_i $(i = 1, \ldots, n)$:

1) Construct its estimated behavior table from the behavior tables of the basic services.
2) Calculate the behavior similarity $\Phi(s, t_i)$.

Step 4: Let t denote the composite service t_i for which the behavior similarity is highest. Return t.

This procedure describes how to find the optimal repair. Pragmatically, our method applies this procedure to derive a *ranked list* of repairs that are recommended to the programmer. This provides the programmer the opportunity to apply his judgment. In the previous example, the output might look like

1) $s_0 \longrightarrow s_{11}, s_{46}, s_{23}$
2) $(s_1, s_2) \circ s_0 \longrightarrow s_{31}, s_{18}, s_{123}$
3) $s_3 \circ (s_1, s_2) \circ s_0 \longrightarrow s_{312}, s_{12}, s_{65}$

where for each sub-expression a list of three repairs are proposed, in order of similarity.

V. Experimentation

Two aspects of the repair procedure need validation so an extensive experiment was designed for each. The first aspect concerns the quality of an intermediate construct used in the procedure, and the second aspect concerns the quality of the suggested substitutions.

A. Is the composition of estimates of behavior a good estimate for the behavior of the composition?

In Step 2.1 and in Step 3.1, the repair procedure estimates the behavior table of a composite service, from estimated behavior tables of its component services (Equations 6 and 7). How good are such estimates? Our experiment is as follows:

1) Create n basic services in each of two classes, where each service in the first class can be composed with each service in the second class; e.g., n services of the type (A, B) and n services of the type (B, C). Each of these services has a considerable number of instances; we assume the number of instances in each of the services is m.[9] All samples (estimated behaviors) were constructed with k instances each.

[7]These are available from the service repository $\mathcal{R}$.

[8]These require the behavior of α_i which was estimated in the previous step, and the behavior of s_i which are available from the repository.

[9]The two classes need not have the same number of services and the number of instances in each service need not be the same.

2) Compose each service in the first class with each service in the second class, for a total of n^2 compositions (note that each composition has m instances).
3) Sample the behavior of each of the basic $2n$ services, and each of the n^2 composite services.
4) Compose each of the n behavior samples in the first class with each of the behavior samples in the second class, for a total of n^2 samples of the compositions.
5) Compare the n^2 compositions of samples with the n^2 samples of compositions.

Clearly, a sample of the composition of two behaviors will generally be different from the composition of two samples of the same two behaviors. But we measure the quality of the method by whether the two sets of n^2 behavior samples *cluster* similarly. We partition each set into an equal number of clusters (we use $\sqrt{n}$ clusters). We now check how *similar* are the clusterings. This is done by calculating the proportion of situations in which two samples that are in the same cluster in one clustering — are also in the same cluster in the other clustering.

To test the robustness of the method, two statistical properties of the tables, called *composability* q and *range* r, were also varied. Note that the table (A, B) is always keyed on its A column, and the table (B, C) is always keyed on its B column. First, assuming the cardinality of the domain of B is p, the *composability* of the two services is defined as the ratio $q = m/p$. Composability is therefore the probability that a random instance of the first table will find a match in the second table. When $q = 1$ the second table covers the entire domain B and hence all instances in the first table find a match in the second table. Second, intuitively, clustering of the compositions (A, C) translates to clustering of sequences of length m of values from the domain of C. Such clustering is affected by the size of the domain, which we denote r. Obviously, narrow ranges create fewer clusters.

The experiment adopted initial values $n = 25$, $m = 1,000$, $k = 10$, $q = 1$ and $r = 10$; that is, we defined 25 services of type (A, B) and 25 services of type (B, C). Each service was described by a behavior table of 1,000 instances and an estimated behavior table of size 10 (1% sample). The composability was perfect and the number of different C values was set to 10.

The number of compositions was therefore $n^2 = 525$. These 525 compositions were clustered twice: by the estimates of the compositions (this could be considered the "actual") and by the compositions of the estimates (our method).

To measure the agreement between the two clusterings, we observed the $525 \cdot 525 = 275,625$ possible *pairs* of compositions; when a pair fell into the same cluster in both clusterings it was scored a *success*. Clustering agreement was then defined as the proportion of successes. This entire experiment was repeated 30 times and the agreement scores were averaged. The final score was 40%. To interpret this score, we compared the first clustering with a random clustering. This experiment was also repeated 30 times and the agreement scores were averaged. The final score was only 22%. That is, our quick method of obtaining behavior samples of composite services performs 80% better than random.

We then repeated the experiment with different values for m, k, r and q. The results are tabulated below:

m	100	41%
	500	41%
	5,000	35%
k	2	26%
	4	32%
	8	37%
	16	46%
	32	53%
r	100	40%
	1,000	39%
	10,000	42%
q	0.5	39%
	0.2	41%
	0.1	40%
	0.05	36%

As can be observed, varying the value of m, r and q had little impact on the cluster agreement scores, implying that the method is not sensitive to the size of the behavior tables or to their statistical properties. However, the method seems to correlate positively with the size of the sample, which is rather to be expected.

B. Are the Suggested Repairs of High Quality?

In Steps 2.3 and in Step 4 the repair procedure finds the "best" substitution for a sub-expression. In Step 2.3 it is a repository service that substitutes a sub-expression that incorporates the failed component; In Step 4 it is a newly constructed composite service that substitutes the original composite service. Our second experiment validates that the procedure indeed suggests high quality substitutions.

In this experiment, again, we create two classes of composable services. However, the n services in each class are generated so that they fall into $\sqrt{n}$ subsets, each with $\sqrt{n}$ services, where the services in each subset *behave similarly*.[10] We then compose each service in the first class with each service in the second class for a total of n^2 compositions. Since each class has been divided into $\sqrt{n}$ subsets of similar semantics, the n^2 compositions are now divided into n subsets of n similar services.

We now choose at random a service c from the group of n^2 composite services, we compare this service with the other $n^2 - 1$ composite services, and we select the service c' with the highest behavior similarity Φ. If c' and c are in the same behavior group it is scored a *success*.

This experiment adopted the same values of $n = 25$, $m = 1,000$, $k = 10$, $q = 1$ and $r = 10$; and was repeated 30 times. The overall ratio of success was 17%; that is, in only 17% of the cases, our metric for behavior similarity selected a service of similar semantics. However, the rate of success

[10]The two classes need not be partitioned into the same number of subsets, and each subset need not have the same number of similar services.

increased to 35% when a success was defined "at least one of the 3 closest neighbors of c is in the same behavior group as c". With 5 closest neighbors it increased to 53%, and with 10 closest neighbors it increased to 85%. To summarize, when a composition of two services is to be replaced, in 85% of the time a proper substitution is included among the top 10 suggestions. Recall that there could be 524 suggestions, and a random choice of 10 services would have a chance of only about 33% to include one from the similarity group of c.

VI. Conclusion

We addressed the issue of failure in service compositions, and proposed a method for recommending service repairs. For each portion of the composition that incorporated the failed service, a list of possible substitutions was suggested in order of similarity to the original composition. Our work is based on a minimalistic model, which could serve as a basis for a more elaborate system that would incorporate additional features and functionalities.

The work is ongoing and we sketch here five of the issues that are under consideration. Our repair procedure performs an exhaustive search for the optimal solution. In the case of large service repositories, such a search could become expensive. Research is underway to discover heuristics that would reduce the search space. Since repository services could evolve there should be a mechanism that would allow periodic updates of stored behaviors. Our similarity measure is based on behavior only. An obvious extension is to incorporate quality of service parameters, such as reliability or response time, into the definition of similarity. Our experiments used synthetic services. In the future we would like to test our methodology with "real-world" services. The behavior of "real-world" services could be extracted from deployment logs or other histories of program executions. Finally, our methodology is intended to assist programmers in maintaining the operability of service compositions. Towards this goal, it would be beneficial to conduct a user study, in which an implementation of our methodology would be available to teams of actual users.

References

[1] J. Andersson, R. De Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. *Software Engineering for Self-Adaptive Systems*, pages 27–47, 2009.

[2] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, pages 369–384, 2007.

[3] Sushil K Bajracharya, Joel Ossher, and Cristina V Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 157–166, 2010.

[4] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 237–246, 2010.

[5] Joshua Church and Amihai Motro. Learning service behavior with progressive testing. In *4th IEEE International Conference on Services Oriented Computing and Applications (SOCA)*, 2011.

[6] Joshua Church and Amihai Motro. Learning service behavior with progressive testing. In *Proceedings of SOCA 11, IEEE International Conference on Service-Oriented Computing and Applications*, pages 1–8, 2011.

[7] Joshua Church and Amihai Motro. Efficient service substitutions with behavior based metrics. In *20th IEEE International Conference on Web Services (ICWS)*, 2013.

[8] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351, 2005.

[9] Sarah Cohen-Boulakia and Ulf Leser. Search, adapt, and reuse: the future of scientific workflows. *ACM SIGMOD Record*, 40(2):6–16, 2011.

[10] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *Proceedings of VLDB-2004, Thirtieth International Conference on Very Large Data Bases*, pages 372–383, 2004.

[11] Tiago Espinha, Andy Zaidman, and H-G Gross. Understanding service-oriented systems using dynamic analysis. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the*, pages 1–5, 2011.

[12] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of ICSE 2010, ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 475–484, 2010.

[13] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. Applying classification techniques to remotely-collected program execution data. *ACM SIGSOFT Software Engineering Notes*, 30(5):146–155, 2005.

[14] Ahmed E. Hassan. The road ahead for mining software repositories. In *FoSM 2008, Frontiers of Software Maintenance*, pages 48–57, 2008.

[15] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 70–79, 2009.

[16] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 215–224, 2010.

[17] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92, 2009.

[18] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[19] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[20] Iulian Neamtiu and Tudor Dumitras. Cloud software upgrades: Challenges and opportunities. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the*, pages 1–10, 2011.

[21] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781, 2013.

[22] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 245–256, 2009.

[23] Mícheál O'Searcoid. *Metric spaces*. Springer, 2006.

[24] Atousa Pahlevan, Sean Chester, Alex Thomo, and HA Muller. On supporting dynamic web service selection with histogramming. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the*, pages 1–8, 2011.

[25] Steven P Reiss. Semantics-based code search. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 243–253, 2009.

[26] Q. Yu and A. Bouguettaya. Framework for web service query algebra and optimization. *ACM Transactions on the Web (TWEB)*, 2(1):1–35, 2008.

[27] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6, 2007.

[28] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity search*. Springer, 2006.